



AFRL-RI-RS-TM-2020-001
VERSION 2 of 4

EDGE OF THE ART IN VULNERABILITY RESEARCH VERSION 2 OF 4

TWO SIX LABS

JANUARY 2021

TECHNICAL MEMORANDUM

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

NOTICE AND SIGNATURE PAGE

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

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

AFRL-RI-RS-TM-2020-001 Version 2 of 4 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

AMANDA OZANAM
Work Unit Manager

/ S /

JAMES PERRETTA
Deputy Chief, Information
Exploitation & Operations Division
Information Directorate

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

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) JANUARY 2021		2. REPORT TYPE TECHNICAL MEMORANDUM		3. DATES COVERED (From - To) SEP 2019 - JAN 2020	
4. TITLE AND SUBTITLE Edge of the Art in Vulnerability Research Version 2 of 4				5a. CONTRACT NUMBER FA8750-19-C-0009	
				5b. GRANT NUMBER N/A	
				5c. PROGRAM ELEMENT NUMBER 62303E	
6. AUTHOR(S) Scott Tenaglia Perri Adams				5d. PROJECT NUMBER CHES	
				5e. TASK NUMBER S4	
				5f. WORK UNIT NUMBER 01	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Two Six Labs 901 N Stuart Street, Suite 1000 Arlington, VA 22203				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/RIGA 3701 North Fairfax Drive 525 Brooks Road Arlington, VA 22203-1714 Rome, NY 13441-4505				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TM-2020-001 Version 2 of 4	
12. DISTRIBUTION AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited. DARPA DISTAR CASE # 32474 Date Cleared: April 8, 2020					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT This Edge of the Art report aggregates the most recent advances in vulnerability research (VR), reverse engineering (RE), and program analysis tools and techniques that Two Six Labs considers when planning for the next CHES evaluation event.					
15. SUBJECT TERMS Vulnerability Research, Reverse Engineering, Program Analysis, Cyber, Fuzzing, Software Security					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON AMANDA OZANAM
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER (Include area code)
U	U	U	UU	81	N/A

TABLE OF CONTENTS

1	INTRODUCTION	2
2	SCOPE OF THE DOCUMENT	3
2.1	TOOLS CRITERIA	3
2.2	TECHNIQUE CRITERIA	3
2.3	TOOL AND TECHNIQUE CATEGORIES	4
3	STATIC ANALYSIS AND INTERMEDIATE REPRESENTATIONS.....	4
3.1	TECHNICAL OVERVIEW	4
3.2	INTERMEDIATE REPRESENTATIONS	11
3.2.1	<i>Miasm</i>	11
3.2.2	<i>BAP</i>	15
3.2.3	<i>GTIRB</i>	19
3.3	STATIC INSTRUMENTATION	21
3.3.1	<i>Retrowrite</i>	21
3.4	TRENDS	23
4	DYNAMIC ANALYSIS AND SYMBOLIC EXECUTION	24
4.1	TECHNICAL OVERVIEW	24
4.2	DEBUGGERS	29
4.2.1	<i>drgn</i>	29
4.3	DYNAMIC BINARY INSTRUMENTATION	30
4.3.1	<i>Qiling</i>	30
4.4	DYNAMIC AND SYMBOLIC BINARY ANALYSIS	31
4.4.1	<i>Triton</i>	31
4.5	CONSTRAINT SOLVERS	34
4.5.1	<i>Just Fuzz It Solver</i>	34
4.6	TRENDS	36
5	FUZZING	37
5.1	TECHNICAL OVERVIEW	37
5.2	MUTATION-BASED FUZZING	39
5.2.1	<i>honggfuzz</i>	39
5.2.2	<i>Frizzer</i>	41
5.2.3	<i>Redqueen</i>	43
5.3	GENERATION-BASED FUZZING.....	44
5.3.1	<i>F1 Fuzzer</i>	44
5.4	PLATFORM SPECIFIC FUZZING	47
5.4.1	<i>kAFL</i>	47
5.5	FRAMEWORKS	49
5.5.1	<i>Enfuzz</i>	49
5.6	TRIAGE	52
5.6.1	<i>Halfempty</i>	52
5.7	TRENDS	53
6	EXPLOITATION.....	55
6.1	TECHNICAL OVERVIEW	55
6.2	HEAP	58
6.2.1	<i>SIEVE and SHRIKE</i>	58
6.2.2	<i>HeapHopper</i>	62
6.2.3	<i>shadow</i>	64
6.3	SHELLCODE	66
6.3.1	<i>SynesthesiaYS</i>	66
6.4	TRENDS	68
7	CONCLUSION	69
8	REFERENCES.....	70



1 Introduction

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

Combining Hacker Expertise Can Krush Machine Assisted Target Exploitation (CHECKMATE), the 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 against which the TA1 and TA2 performers will be measured. Vulnerability research is a constantly evolving area of cyber security, which means that the baseline for measuring the success of the CHES program is a moving target. The control team must keep pace with the most recent advancements to remain an effective baseline for comparison. The CHECKMATE team not only needs to stay on top of the state-of-the-art research and technology solutions, but also capture the most emerging and trending techniques across all relevant vulnerability classes, tools, and methodologies. This *Edge of the Art* report aggregates the most recent advances in vulnerability research (VR), reverse engineering (RE), and program analysis tools and techniques that the CHECKMATE team considers when planning for the next CHES evaluation event.

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

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

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



2 Scope of the Document

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

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

2.1 Tools Criteria

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

Year Released – “Cutting edge” has an obvious temporal component, but it is less obvious where the cut-off should lie. Every tool in this report has been introduced within the last five years (i.e. first released in 2014 or later). Over two-thirds of the tools were released in or after 2017, and most of those were released in or after 2018. Those released earlier are included because they have significantly matured since their initial release and now contain notable features.

Capability – New tool capabilities, and how they compare to the current state-of-the-art, are a primary consideration for inclusion in this report. The novel aspect of a new tool 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.

2.2 Technique Criteria



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

2.3 Tool and Technique Categories

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

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

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

3 Static Analysis and Intermediate Representations

3.1 Technical Overview

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

Disassembly

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



An assembler converts a program from assembly language to machine code, and a disassembler performs the reverse: it converts machine code to assembly language. Since there is often a one-to-one correspondence between machine instructions and assembly instructions, this translation is much less complicated than decompilation. However, disassembly can pose challenges, especially with architectures like x86 which have variable length instructions. When overlapping sequences of bytes could themselves be valid instructions, 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) [1, p. 2]. Many popular disassemblers including IDA Pro [2] and Binary Ninja [3] use the latter technique.

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

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

Reassembleable Disassembly

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

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



Static Binary Rewriting and Static Instrumentation

Relevant EotA Tools: Retrowrite, LIEF

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

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

An intermediate representation is a form of the program that is in-between both its source language and target architecture representations. The semantics and human intelligibility of an IR often fall between the source language and target architecture. IRs may be expressed using a variety of formats, however most often they take the form of 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 [15], the IR used in the Clang compiler [16]. LLVM is helpful as an example not just because it is well known, but because it demonstrates the range of features a well-designed IR can offer. The instruction set and type system for LLVM is language independent, which means there are no high-level types and attributes. This allows LLVM to be ported to many architectures. Although the type system is low-level, by providing type information, LLVM enables a target program to be optimized through various analyses [17]. Unlike machine code however, LLVM is designed to be human readable [17].

LLVM uses a technique called Single Static Assignment (SSA), which is 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 [18]. The decompiler Ghidra does something similar but in reverse: a binary program is first lifted (converted to a higher-level representation) to an IR called P-Code [19], on which Ghidra can perform analyses and then decompile by converting the program to pseudo source code. Therefore, Ghidra can decompile anything it can lift to P-Code, because decompilation is performed on a language agnostic IR and not the original machine language [20].

Ghidra uses SSA form in its decompilation, however unlike LLVM, P-Code is not in SSA form by default [20]. 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 [3].

SSA demonstrates one of the trade-offs that inform IR design. The developers of Binary Ninja created the charts in Figure 1 and Figure 2 to show the tension between different features of IRs.

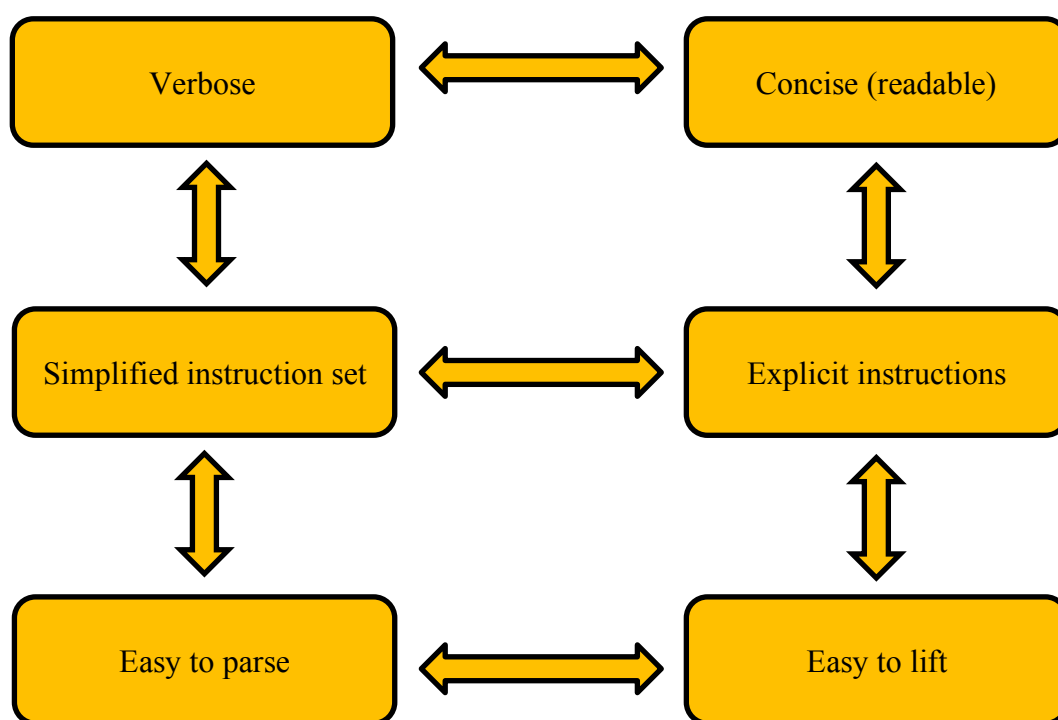


Figure 1: Tradeoffs of IRs, Pt. 1 [21, p. 29]

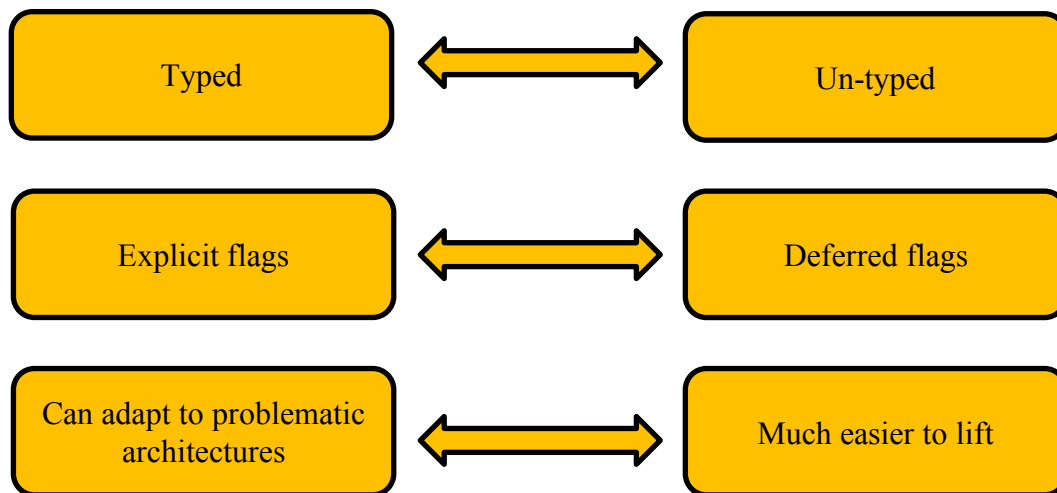


Figure 2: Tradeoffs of IRs, Pt. 2 [21, p. 30]

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

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

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

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 [29]. WinAFL, a version of the AFL fuzzer for the the Windows operating system, uses DynamoRIO, a dynamic binary instrumentation engine with its own IR [30].

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

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

Decompilation

Relevant EotA Tools: IDA Pro, Ghidra

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

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

As of 2019, the United States National Security Agency (NSA) released Ghidra, a disassembler and decompiler with comparable performance to IDA Pro [34]. Each has its respective strengths and weaknesses, however, for certain constructs like jump tables and no return functions, IDA clearly performs better. Binary Ninja is also developing a decompiler which is expected to be released soon.

Static Vulnerability Discovery

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

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) [29, p. 4].
- **Variable and Type Information Recovery:** Variable and type information is used by the compiler but is not present in final binary executable form (unless the binary is compiled to explicitly include this information for debugging purposes). Therefore, it is often necessary to recover this information when analyzing a binary [1, p. 5]. One attribute of many IRs is that their lifters will recover variable and type information and include it in the IR 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 [1, p. 5].
- **Value Set Analysis (VSA):** VSA is a form of static analysis which attempts to track values and references throughout a binary [29, p. 5]. This analysis has a variety of uses, including identifying indirect jumps or find vulnerabilities like out of bound accesses.
- **Graph-based vulnerability discovery:** This form applies graph analysis to a CFG to identify vulnerabilities [29, p. 5].
- **Symbolic Execution:** Symbolic execution replaces program inputs with symbolic values, and then symbolically executes over the program. Symbolic execution be done either statically or dynamically, and it is discussed further in Section 4.
- **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 for these forms of analysis, including in a recent update [38].



3.2 Intermediate Representations

3.2.1 Miasm

Reference Link	https://miasm.re
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	X86 (32, 64); ARM (32, 64); PowerPC (32, 64)
Initial Release	First Commit: 2014 Initial Release: March 12, 2018
License Type	Open-Source
Maintenance	Maintained by Carnegie Mellon University (CMU)

Overview

Miasm is a reverse engineering framework developed by the IT Security division of the French Alternative Energy and Atomic Energy Commission (CEA) [27]. Miasm is based on a custom Intermediate Representation (IR), Miasm IR [26]. Statically, Miasm can modify binaries, assemble and disassemble a range of architectures, and lift to their IR. The Miasm IR enables many of Miasm's capabilities, including deobfuscation and symbolic execution. It is designed to easily track reads and writes to registers and memory. Recently, Miasm improved their IR by adding, among other features, SSA transformations and high-level flag-operators [39].

Miasm has a built-in emulation engine, which converts Miasm IR to machine instruction using Just-In-Time (JIT) compilation [27]. This enables another built in feature of Miasm, the symbolic execution engine. Additionally, there is now a concolic execution engine which allows for symbolic tracing of the execution of a concrete value [1].

Design and Implementation

Miasm is written in Python and distributed as a Python library. The following code snippet is a simple example of using Miasm to assemble and disassemble a single instruction [27].



```

1  from miasm.arch.x86.arch import mn_x86
2  from miasm.arch.x86.regs import EDX
3  from miasm.core.locationdb import LocationDB
4
5  # Create a location database
6  loc_db = LocationDB()
7
8  # Create an instruction
9  instruction = mn_x86.fromstring('MOV EAX, EBX', loc_db, 32)
10
11 # Modify an operand: MOV EAX, EBX -> 'MOV EDX, EBX'
12 instruction.args[0] = EDX
13
14 # Assemble an instruction
15 assembly = mn_x86.asm(instruction)
16
17 # Disassemble an instruction
18 instruction = mn_x86.dis(assembly, 32)

```

Figure 3: Example usage of Miasm [27]

Miasm IR: Miasm IR is built with nine types of expressions, shown in Table 1.



Expression Type	Description	Example	
ExprInt (value, size)	Integer	0xC	ExprInt (0xC, 32)
ExprMem (address, size)	Memory Location, dereferenced from the address given Can be a left value e.g., @32[ESI] = EBP	@32[ESI]	ExprMem (@ [ExprId ('ESI' , 32) , 32]
ExprId (name, size)	Variable Can be a left value e.g., EAX = EBP	EAX	ExprId ('EAX' , 32)
ExprLoc (location, size)	Location	loc_17	ExprLoc (loc_17, 32)
ExprCond (condition, value_if_true, value_if_false)	Ternary Expression	EAX ? EBX : ECX	ExprCond (ExprId ('EAX' , 32) , ExprId ('EBX' , 32) , ExprId ('ECX' , 32))
ExprOp (operator_name, operand1, operand2)	Assignment, this is the only expression which modifies the register or memory state.	EAX + EBX	ExprOp ('+' , ExprId ('EAX' , 32) , ExprId ('EBX' , 32))
ExprAssign (destination, source)	Operation	EAX = EBX	ExprAssign (ExprId ('EAX' , 32) , ExprId ('EBX' , 32))
ExprSlice (source, bit_start, bit_stop)	Assignment, this is the only expression which modifies the register or memory state.	AH = EAX[8:16]	ExprSlice (ExprId ('EAX' , 32) , 8, 32)
ExprCompose (expression1, expression2)	Slice of bits extracted from a given expression	AX = AH.AL	ExprCompose (ExprId ('AH' , 8) , ExprId ('AL' , 8)

Table 1: Expression types in Miasm IR , [40, pp. 12-14] [26, pp. 6-7], [27, p. "miasm/miasm/expression/expression.py"]

As shown in Table 1, Miasm IR is like an abstract syntax tree. It is easy to track reads and writes throughout a program because the assignment operator, ExprAssign, is the only expression that manipulates the memory state of the program [26, p. 6]. This is useful in deobfuscation and optimization.

Figure 4 and Figure 5 is an example of this translation on the first six instructions of the main function in a sample binary. Figure 4 is the original x86 disassembly, and Figure 5 is the resulting Miasm IR. The example Miasm IR in Figure 5 shows how Miasm IR represents the updated register content and resulting flags from relevant operations.



```

push ebp
mov ebp, esp
and esp, 0xFFFFFFFF0
sub esp, 0x20
mov DWORD PTR [esp], 0x80487D0
call 8048560 <puts@plt>

```

Figure 4: Example excerpt of an x86 ELF [41]

```

ESP = ESP + -0x4
@32[ESP + -0x4] = EBP

EBP = ESP

zf = FLAG_EQ_AND(ESP, 0xFFFFFFFF0)
nf = FLAG_SIGN_SUB(ESP & 0xFFFFFFFF0, 0x0)
pf = parity((ESP & 0xFFFFFFFF0) & 0xFF)
of = 0x0
cf = 0x0
ESP = ESP & 0xFFFFFFFF0

zf = FLAG_EQ_CMP(ESP, 0x20)
nf = FLAG_SIGN_SUB(ESP, 0x20)
pf = parity((ESP + -0x20) & 0xFF)
cf = FLAG_SUB_CF(ESP, 0x20)
of = FLAG_SUB_OF(ESP, 0x20)
af = ((ESP ^ 0x20) ^ (ESP + -0x20))[4:5]
ESP = ESP + -0x20

@32[ESP] = 0x80487D0

EAX = call_func_ret(loc_8048560, ESP)
ESP = call_func_stack(loc_8048560, ESP)

IRDst = loc_80486c2

```

Figure 5: Miasm IR for Figure 4

For an assembly instruction that alters flag values, such as the x86 instruction `cmp eax, ebx`, Miasm was recently updated to use high-level operators. Whereas the ‘cf’ flag after this instruction used to be calculated as follows,

$$cf = (((EAX \wedge EBX) \wedge (EAX + -EBX)) \wedge ((EAX \wedge (EAX + -EBX)) \wedge (EAX \wedge EBX)))[31:32]$$

it is now

$$cf = FLAG_SUB_CF(EAX, EBX) \quad [42].$$


Also, like Binary Ninja, Miasm now offers an SSA transformation that will convert their IR into SSA form [39].

Emulation: Miasm can perform emulation using a custom Just-In-Time (JIT) engine which converts Miasm IR with either LLVM, GCC, Clang, or Python [27].

Symbolic Execution and Dynamic Symbolic Execution: Miasm has a built-in symbolic execution engine, which can be called using its Python libraries. Additionally, a feature that would allow the symbolic tracing of concrete inputs was added in April 2017 [43]. Miasm IR already offered the ability to perform this form of concolic execution, however this update provided an implementation of it, called `DSEEngine` [43].

Use Cases and Limitations

Miasm IR has been a useful addition to the numerous IRs and IR frameworks. Miasm IR offers a way to track reads and writes to registers and memory, because it is expression-based and the program state can only be modified through the `ExprAssign` expression [26, p. 5]. This contrasts with most other IRs discussed in this report, which typically modify the program state with statements like calls, in addition to assignment statements.

Miasm IR also makes Miasm useful as a deobfuscation and simplification tool. Deobfuscation is a large use case for Miasm because it offers integration with Windows and a suite of IDA Plugins [27].

3.2.2 BAP

Reference Link	https://github.com/BinaryAnalysisPlatform/bap
Target Type	Binary
Host Operating System	Linux
Target Operating System	Linux; Windows
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64); ARM (32, 64); PowerPC (32, 64)
Initial Release	2009 v1.0.0: September 21, 2016 v2.0.0: November 12, 2019
License Type	Open-Source
Maintenance	Maintained by Carnegie Mellon University (CMU)

Overview

BAP (Binary Analysis Platform) is a binary analysis framework built around a custom Intermediate Representation (IR) called BAP Intermediate Language (BIL). As described in the repository [24] and the documentation [44], BAP is written in OCaml and is a command-line tool for which custom plugins can be written. It also offers an IDAPlugin. BAP disassembles binaries and then lifts them to BIL and uses this to build a control flow graph. These representations can



be used for different user-defined analyses or as instrumentation to perform symbolic execution, among other things. BAP is used as the basis for the symbolic execution engine Mayhem, which placed first in DARPA's Cyber Grand Challenge (CGC). Although BAP has been available for over a decade [45], they released a new version in November 2019 [38].

Design and Implementation

BAP is a library written in OCaml, which includes an LLVM-based disassembler and a custom lifter and can be leveraged with user-created plug-ins written in OCaml or Python. BAP decomposes a program into "Terms," such as a basic block, and those terms are tagged with attributes, such as the instructions within that block. In BAP, analyses are termed "Passes."

BAP Intermediate Language (BIL): BAP first translates a binary's assembly to the BIL IR, which is used in subsequent analysis. As described in the formal specification [25], BAP uses constructs common in functional programming languages (e.g., `let` bindings) and it models side effects explicitly (e.g., modifications to flags or the stack). BIL is not SSA form but BAP allows the user to translate BIL to an equivalent SSA form.

BAP comes with a verbose formal specification. A high-level understanding of the language can be understood from the following excerpt from [25] shown in Figure 6.



BIL program is represented as a sequence of statements. Each statement performs some side-effectful computation.

```

bil, seq    ::=
    | {s1; ..; sn} S

stmt, s     ::=
    | var := exp                - assign exp to var
    | jmp e                     - transfer control to a given address e
    | cpuexn (num)              - interrupt CPU with a given interrupt num
    | special (string)          - instruction with unknown semantics
    | while (exp) seq           - eval seq while exp is true
    | if (e) seq                 S - eval seq if e is true
    | if (e) seq1 else seq2    - if e is true then eval seq1 else seq2

```

BIL expressions are side-effect free. Expressions include a usual set of operations on bitvectors, like arithmetic operations and converting bitvectors of one size to bitvectors of another size (casting in BIL parlance).

```

exp, e      ::=
    | (exp)                      S
    | var                        - a variable
    | word                       - an immediate value
    | e1[e2, endian] : nat       - load a value from address e2 at storage e1
    | e1 with [e2, endian] : nat ← e3 - update a storage e1 with binding e2 ← e3
    | e1 bop e2                - perform binary operation on e1 and e2
    | uop e1                    - perform an unary operation on e1
    | cast : nat[e]              - extract or extend bitvector e
    | let var = e1 in e2        - bind e1 to var in expression e2
    | unknown [string] : type    - unknown or undefined value of a given type
    | if e1 then e2 else e3    - evaluates to e2 if e1 is true else to e3
    | extract : nat1 : nat2[e]  - extract or extend bitvector e
    | e1@e2                    - concatenate two bitvector e1 to e2

```

Figure 6: Excerpt from the BIL formal specification [25, p. 2]

Figure 7 and Figure 8 is an example of this translation on the first six instructions of the main function in a sample binary. Figure 7 is the original x86 disassembly, and Figure 8 is the resulting BIL. The example BIL in Figure 8 shows how BIL represents the updated register content and resulting BIL from relevant operations.

```

push ebp
mov  ebp, esp
and  esp, 0xFFFFFFFF0
sub  esp, 0x20
mov  DWORD PTR [esp], 0x80487D0
call 8048560 <puts@plt>

```

Figure 7: Example excerpt of an x86 ELF [41]



User Plugins: User defined plugins are a key feature of BAP. The user can define analyses which will be applied to the disassembly in passes before the final lifting to BIL. The user can also use a scripting language to perform dynamic operations, and emulate the code using Primus, the emulation and micro-execution engine.

```

0x00000bd4: sub main(main_argc, main_argv, main_result)
0x00000bf4: main_argc :: in u32 = mem[ESP + 4, el]:u32
0x00000bf5: main_argv :: in out u32 = mem[ESP + 8, el]:u32
0x00000bf6: main_result :: out u32 = EAX
0x0000061c:
0x00000623: #77 := low:32[EBP]
0x00000626: ESP := ESP - 4
0x00000629: mem := mem with [ESP, el]:u32 <- #77
0x00000630: EBP := ESP
0x0000063d: ESP := low:32[ESP] & 0xFFFFFFFF0
0x00000640: OF := 0
0x00000643: CF := 0
0x00000646: AF := unknown[bits]:u1
0x00000649: PF := ~low:1[let $1 = low:32[ESP] >> 4 ^ low:32[ESP] in
let $2 = $1 >> 2 ^ $1 in $2 >> 1 ^ $2]
0x0000064c: SF := high:1[low:32[ESP]]
0x0000064f: ZF := 0 = low:32[ESP]
0x0000065d: #79 := low:32[ESP]
0x00000660: ESP := low:32[ESP] - 0x20
0x00000663: CF := #79 < 0x20
0x00000666: OF := high:1[(#79 ^ 0x20) & (#79 ^ low:32[ESP])]
0x00000669: AF := 0x10 = (0x10 & (low:32[ESP] ^ #79 ^ 0x20))
0x0000066c: PF := ~low:1[let $1 = low:32[ESP] >> 4 ^ low:32[ESP] in
let $2 = $1 >> 2 ^ $1 in $2 >> 1 ^ $2]
0x0000066f: SF := high:1[low:32[ESP]]
0x00000672: ZF := 0 = low:32[ESP]
0x00000679: mem := mem with [ESP, el]:u32 <- 0x80487D0
0x00000682: ESP := ESP - 4
0x00000685: mem := mem with [ESP, el]:u32 <- 0x80486C2
0x00000688: call @puts with return %0000068a

```

Figure 8: BIL translation of Figure 7

Symbolic Execution: Using user defined plug-ins, and Primus (the emulation engine), a user can use BAP to perform symbolic execution over a binary. This is one of the intended uses for BAP, however BAP does not have prebuilt functionality for it.

Taint Analysis: Primus can be used to perform taint analysis on a binary. This is done via the *Bap_taint* framework. BAP offers an IDA Pro plug-in, and by combining this and IDAPython, the user can view the results of taint analysis in IDA Pro.

Other Features: BAP also has a string deobfuscator called Beagle, a fast policy checker called Saluki, and WUR, which warns for unused results. BAP can be used for dynamic and static taint analysis, and has prebuilt plugins for ROP and UAF exploitation.



New Features

BAP 2.0 was released on November 12, 2019 and was presented as a major release [38]. The following new features are in addition to general bug fixes and software maintenance, and are described in [38]:

Knowledge Representation and Reasoning System (KB): The KB enables mathematical program analysis to be performed by storing ‘facts’ about the target program. The design of the KB draws heavily on logic programming, specifically on Datalog, but the implementation does not use a deductive database. Rather, object properties are stored and updated in such a way that the resulting structure can be used for certain kinds of program analysis. More specifically, these object properties are modeled by a directed-complete partially ordered set and updated monotonically, which means that facts can only be refined from their previous value. This representation enables a user to collect facts, or object properties, about a binary and then perform operations on that knowledge, such as computing the least fixed point of a property.

Core Theory: Because BIL is non-extensible, BAP now includes Core Theory. This is a new IR, that is extensible and enables the user to add new features to the language without compromising pre-existing analyses. Core Theory can be considered a superset of BIL. Core Theory adds functionality to support modular arithmetic, bit vector operations, memory register manipulation and floating-point math.

New Interface: BAP 2.0 offers significant updates to the BAP command-line tool and improved appearance, especially regarding Control Flow Graph (CFG) output.

Use Cases and Limitations

BAP can be used to perform many kinds of binary analysis including trace analysis and symbolic execution. BAP was designed to perform sophisticated kinds of program analysis which makes it a powerful tool in this regard. However, it also requires a sophisticated understanding of the concepts on which it was built. The documentation for BAP’s APIs is very verbose but there is a lack of examples or high-level overviews which would be useful given BAP’s size and range of capabilities.

3.2.3 GTIRB

Reference Link	https://github.com/GrammaTech/gtirb
Target Type	Intermediate Representation
Host/Target Operating System	Linux; Windows
Host/Target Architecture	x86 (32, 64)
Initial Release	October 29, 2018
License Type	Open-Source
Maintenance	Maintained by GrammaTech



Overview

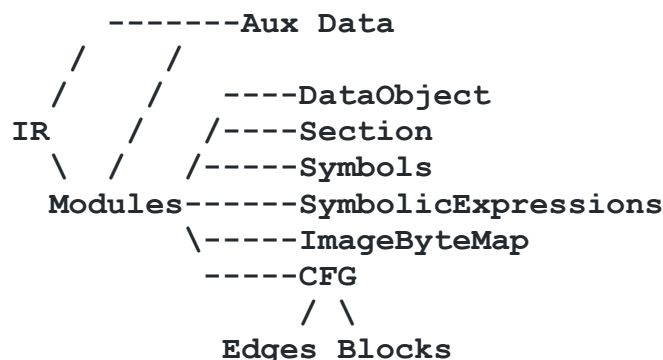
The GrammaTech Intermediate Representation for Binaries (GTIRB) is an IR designed to enable conversion between various other IRs. There are many IRs currently in use in a variety of tooling which can lead to compatibility issues. GTIRB was designed to be a modular solution to facilitate collaboration between multiple IRs. It stores instruction blocks in raw machine-code and has separate data structures for other program information like symbols. This enables GTIRB to be IR-independent, while allowing IR-specific information to be saved as well. [31] [46]

Design and Implementation

GTIRB is a C++ library and has built-in support for Google's Protocol Buffer (Protobuf). It works by separating instructions from other program information and storing the former as raw machine code bytes. The symbolic and control flow information, and optionally certain kinds of auxiliary data like function boundary information or type information, are stored using maps and vectors.

GTIRB can be generated from a binary using GrammaTech's DDisasm disassembler, which is discussed later in this section.

IR: The following is the structure of GTIRB [31].



- **Module:** Loadable Object e.g., an executable or library.
- **DataObject:** Contains data.
- **CFG:** Interprocedural control flow graph, containing basic blocks (**Blocks**) and the control flow edges (**Edges**) between the blocks.
- **ImageByteMap:** Each datum in **DataObject** and each **Block** specify a range into this map [1].
- **Symbol:** Contains symbols; each symbol has a pointer to the **Block** or datum in **DataObject** to which it corresponds.

Each element in the GTIRB (e.g., **Module**, **Symbol**, etc.) has a Universally Unique Identifier (UUID), and the contents of elements can be referred to by the offset from its parent, and the



parent's UUID (e.g., instructions can be referred to by the UUID of their parent **Block** and their offset from its start).

C++ Library: The C++ library is used for populating the GTIRB, serializing and deserializing, and querying the IR.

Protobuf: The user can use Google's Protocol Buffer (Protobuf) to package GTIRB and transmit it between tools.

Use Case and Limitations

The intended use case is to facilitate collaboration between multiple IRs. As this document demonstrates, there are numerous Intermediate Representations currently in use, and tool compatibility is a common issue.

The user is still required to write the conversion to GTIRB format themselves. This format may also pose issues for certain IRs which cannot fit into GTIRB's storage constraints due to their design.

3.3 Static Instrumentation

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

3.3.1 Retrowrite

Reference Link	https://github.com/HexHive/retrowrite
Target Type	Binary (64-bit ELF with PIE enabled)
Host/Target Operating System	Linux
Host/Target Architecture	x86 (64)
Initial Release	June 2019
License Type	Open-Source
Maintenance	Last commit July 2019



Overview

Retrowrite is a static binary rewriter which can instrument binaries for the fuzzer AFL and the memory checker Address Sanitizer (ASAN). The technique was presented in the paper *Retrowrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization* by Sushant Dinesh, Nathan Burow, Dongyan Xu and Mathias Payer [1]. This research could be extended beyond the two applications implemented in the paper, and is based on a custom reassembleable disassembly approach.

Retrowrite enables black-box binaries (i.e., targets without source code) to be run with AFL and Address Sanitizer (ASAN) without needing dynamic binary translation or emulation. In the case of AFL, Retrowrite could be used in place of AFL QEMU mode, which the authors claim it significantly outperforms. Additionally, they assert that binaries instrumented using Retrowrite perform similarly to ones instrumented from source. [1] [47]

Design

Retrowrite relies on a custom static binary rewriting and recompilation technique (recompilation techniques are discussed in Section 3.1). Much of the existing research in this area relies on heuristics for binary rewriting, but when instrumenting a binary for vulnerability discovery, preserving the behavior of the original program is imperative. Therefore, Retrowrite implements a rewriting technique which maintains correctness, but introduces its own limitations.

Retrowrite operates directly on the disassembly itself rather than relying on an Intermediate Representation (their implementation uses Capstone to generate disassembly). However, their technique does rely on Position Independent Code (PIC), which limits the kind of binaries with which this can be used. Their entire process happens over the following five steps described in [1, p. 4]:

1. **Preprocessing:** This step loads the binary, disassemble using linear-sweep, and constructs a “best effort” Control Flow Graph (CFG).
2. **Symbolization:** This is where the reassembleable disassembly is produced. Assembler labels are assigned in three phases to symbolize the references.
 - a. **Control Flow Symbolization:** Addresses which are called or jumped to are assigned labels.
 - b. **PC-relative Addressing:** This part explicitly relies on a feature of PIC, which is that addresses are calculated relative to the Program Counter (PC) (e.g., \$RIP for x86-64). PC-relative addresses are changed to be relative to an assembler label, which is subsequently defined to be the location of PC at that point.
 - c. **Data Relocations:** An address pointed to by a relocation entry is assigned an assembler label, which is then written in the relocation entry, replacing the address.
3. **Instrumentation Pass:** At this point, the binary is instrumented for its intended purpose: the paper implements instrumentation passes for AFL and ASAN. This requires recovering the function information of the original program, as well as statically handling



register allocations for the added instrumentation. This must be done while not breaking the binary and preserving its original behavior.

4. **Instrumentation Optimization:** The side effects of the instrumentation are analyzed to determine which registers can be used for the added instrumentation.
5. **Reassembly:** A reassembleable assembly file is produced.

The Retrowrite technique is currently implemented for two analysis tools, ASAN and AFL. A binary is instrumented for either of these tools, which then produces an assembly file that can be recompiled using gcc, clang or the AFL versions of these compilers. The resulting binary can be run with its intended tool as if it had been instrumented from source code. However, the tool itself has dependency issues and is prone to errors on even stable Ubuntu distributions, which decreases its usability.

Use Cases and Limitations

The paper focuses on two use cases for static binary rewriting: instrumenting for AddressSanitizer (ASAN) and for AFL. At the time of this writing, the existing solutions for performing memory sanitization and for fuzzing binaries rely primarily on dynamic instrumentation and/or emulation, which greatly increases overhead. Retrowrite claims that this technique would allow ASAN and AFL to be run on binary-only programs without significant performance hit or decrease in efficacy. This research can be extended to several applications of static binary rewriting such as instrumentation for fuzzing and sanitization, patching, reassembling, etc.

Retrowrite also implements a new reassembleable assembly approach, which can be repurposed for other applications. However, this relies on information inferred from specific attributes of PIC and thus is limited to 64-bit PIC binaries. Their approach also precludes C++ binaries.

Therefore, a major limitation of Retrowrite is that it only supports 64-bit PIC ELF binaries compiled from C code. Another significant limitation of this work is the tool itself relies on very specific versions of LLVM and other dependencies. This can lead to runtime errors even on the latest stable version of Ubuntu 18.04.

3.4 Trends

Over the last several years, new trends have emerged in the field of static analysis. Among these are static instrumentation, and an increased number and variety of IRs available to users. Both are discussed below.

Static Instrumentation

Traditionally, to fuzz a black-box binary with AFL, users have utilized one of several dynamic instrumentation options, like AFL-QEMU Mode or ALF Unicorn. Recently, there has been research into statically instrumenting black-box binaries, such as Retrowrite. This method is constrained to 64-bit Linux binaries compiled as Position Independent Executables (PIE). However, their research uses, and builds upon current methods in binary rewriting,



reassembleable disassembly, recovery of control flow and variable/type information, and function identification. As static analysis techniques become more sophisticated, so will the ability to statically instrument binaries, which will preclude the need to dynamically emulate them, which is often slow and cumbersome.

Variety of Intermediate Representation

There has been a relatively recent increase in the number and variety of available IRs. These include Binary Ninja's family of IRs, which consist of a low, medium, and a yet to be released, high IR. All have an optional SSA form. IDA has now publicized the API to their IR, and with the release of Ghidra came P-Code. Although less recent, Miasm is also an interesting case, given that it is an expression-based IR.

However, this growing number of IRs has a downside, especially with regard to tool compatibility. This has provoked another push within the industry towards some semblance of standardization, to encourage more tools to use a common IR like LLVM. This approach is not universally applicable because many custom IRs are designed for a specific use. Tools, such as GTIRB, attempt to provide a solution for IR compatibility by creating a method to convert between various intermediate representations.

4 Dynamic Analysis and Symbolic Execution

4.1 Technical Overview

Whereas static analysis examines a binary without running it, dynamic analysis observes a binary as it executes. Dynamic analysis allows the user to inspect actual runtime information about program state, including register and memory values, 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.

This section discusses debuggers (drgn [48]), instrumentation frameworks (Qiling [7]), and dynamic binary analysis frameworks (Triton [49]). Triton can be used for a range of analyses, including taint analysis and symbolic execution. BAP and Miasm are two analysis frameworks, discussed in Section 3, that are often used for symbolic execution and other dynamic techniques, however, given their heavy dependence on their IR's, they were discussed in the static analysis section. This section also includes constraint solving (JFS [50]), because of its use in symbolic/concolic execution tools.

Debuggers

Relevant EotA Tools: drgn, rr, shadow

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 [51]) to implement its functionality.

Dynamic Binary Instrumentation (DBI)

Relevant EotA Tools: Frida, Qiling

DBI, which underlies many dynamic binary analysis techniques, entails modifying the binary, either before or during execution, often by hooking 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 [52] and DynamoRIO [53], which underlie many of the tools discussed in these reports. Both can be used to drive the Windows fuzzer WinAFL [54], and the dynamic binary analysis tool Triton is built around Intel Pin [49]. 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 [52]. 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 [53]. 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 [8] operates).

Dynamic Fuzzing Instrumentation

Relevant EotA Tools: Frida, Qiling,

Although fuzzing is discussed at length in the next section, fuzzing often requires dynamic binary instrumentation to 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 (Section 5.2.2) uses Frida to instrument it.

Memory Checking

Memory checking, whether to find memory bugs or analyze them is a valuable form of dynamic analysis in vulnerability research. To do this, a program is instrumented such that if a memory error is triggered during runtime (e.g., an out of bounds access, Null dereference, or Segmentation Fault) it will be recorded, along with additional contextual information. Several tools exist to do this, such as Valgrind [55], Dr. Memory (a part of the DynamoRIO framework) and LLVM’s Sanitizer Suite which includes Address Sanitizer (ASAN) [15].



Dynamic Taint Analysis

Relevant EotA Tools: Triton, angr

Dynamic taint analysis is a form of dynamic binary analysis 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 just useful for tracking values throughout a program, but identifying which instructions are not affected by user input, which can be used for concolic execution. Triton is particularly effective at dynamic taint analysis and discussed in this section.

Symbolic and Concolic Execution

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

Symbolic analysis is a 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 [56, p. 1]”. This does not mean that only one path is followed, just that a distinct formula is generated for each path. When a branch condition is reached, and both branches are feasible, execution will “fork” and follow both possible paths [56, p. 3]. In the paper *(State of) The Art of War: Offensive Techniques in Binary Analysis* [29], Shoshitaishvili et al. describe this kind of DSE:

“Dynamic symbolic execution, a subset of symbolic execution, is a dynamic technique in the sense that it executes a program in an emulated environment. However, this execution occurs in the *abstract* domain of *symbolic variables*.

...

“Unlike fuzzing, dynamic symbolic execution has an extremely high semantic insight into the target application: such techniques can reason about how to trigger specific desired program states by using the accumulated path constraints to retroactively produce a proper input to the application when one of the paths being executed has triggered a



condition in which the analysis is interested. This makes it an extremely powerful tool in identifying bugs in software and, as a result, dynamic symbolic execution is a very active area of research. [29, p. 6]”

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

The other common definition of DSE and concolic execution refers solely to symbolic execution which is “driven by a specific concrete execution [57, 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 [43], [57, pp. 5-6]. After doing this, additional paths can be explored by negating one (or more) of the collected branch conditions for the path of the concrete input, and then solve for the new path with these negated conditions using an SMT solver in order to generate a new input [57, 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 [58] (a fuzzer discussed in the previous report) is an example of hybrid fuzzing.

There are many tools for symbolic execution, including Triton and Miasm (Section 4.4.1 and 3.2.1, respectively). The tool angr [6] (discussed in the previous report) is one of the best [59], 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 [57]. Additionally, constraint solving is a difficult and time-consuming task. As such, symbolic execution is in many cases not a feasible option or must be constrained to a small area of the program.

Constraint Solving

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

Symbolic execution relies on the ability to solve for the collected path constraints, which is a challenging problem. These constraints can be modeled by satisfiability modulo theories (SMT) which generalize the boolean satisfiability problem (SAT). SAT is an NP-complete problem 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 a boolean expression made up of boolean variables, and the boolean operators AND, OR and NOT.

SMT solvers use various techniques to find values which satisfy the given SMT formula. In *A Survey of Symbolic Execution Techniques* [57], the authors write that,

“SMT solvers show several distinctive strengths. Their core algorithms are generic, and can handle complex combinations of many individual constraints. They can work incrementally and backtrack as constraints are added or removed, and provide explanations for inconsistencies. Theories can be added and combined in arbitrary ways, e.g., to reason about arrays of strings. Decision procedures do not need to be carried out in isolation: often, they are profitably combined to reduce the amount of time spent in heavier procedures, e.g., by solving linear parts first in a non-linear arithmetic formula. Incomplete procedures are valuable too: complete but expensive procedures get called only when conclusive answers could not be produced. All these factors allows SMT solvers to tackle large problems that no single procedure can solve in isolation.

“In a symbolic executor, constraint solving plays a crucial role in checking the feasibility of a path, generating assignments to symbolic variables, and verifying assertions. Over the years, different solvers have been employed by symbolic executors, depending on the supported theories and the relative performance at the time [57, pp. 25-26].

The authors subsequently discuss the Z3 constraint solver, which is widely considered the “leading solution for SMT solving [57, p. 26].” They write that “Z3 offers cutting-edge performance and supports a large number of theories, including bit-vectors, arrays, quantifiers, uninterpreted functions, linear integer and real arithmetic, and nonlinear arithmetic [57, p. 26].”

One major limitation of SMT solvers, which in turn limits the ability of symbolic execution, are their performance. These solvers are hindered by time complexity and the hardness of determine satisfiability.

Floating-Point Reasoning This report features a new constraint solver, Just Fuzz It Solver (JFS), which uses fuzzing to solve for constraints. It is designed specifically for floating point constraints. Floating-point representations are particularly challenging for SMT solvers. IEEE-754 is the floating-point standard almost universally adopted, however certain aspects of these numbers create complications for constraint solvers. As discussed in the paper *Building Better Bit-Blasting for Floating Point Problems* [60], floating-point arithmetic requires rounding, thus properties which hold true when adding and multiplying integers, do not for floating-point (e.g. addition and multiplication on floating-points is not associative or distributive). Additionally, IEEE 754 defines several classes of special numbers which do not adhere to the typical format, such as numbers to represent positive and negative infinity, positive and negative zero, and subnormal numbers. The range of floating-point numbers is also large, and often far larger than necessary for a specific use case.



4.2 Debuggers

4.2.1 drgn

Reference Link	https://github.com/osandov/drgn
Target Type	Binary – ELF Linux kernel
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	August 2017
License Type	GNU GPL v3
Maintenance	Maintained by Omar Sandoval; Last commit on 12/12/2019

Overview

drgn is a Python library for debugging Linux executables compiled from C code, either interactively or via an API. As described in the documentation [61], drgn was originally developed to debug the running Linux kernel and has been extended to support core dumps (kernel vmcore or userspace) and attaching to running user space processes. The core of drgn is written in C and built during installation. It can be imported into any C program by linking against libdrgn.so. The Python implementation provides a wrapper around many of the functions exported by the shared object. [48] [61]

Design and Implementation

drgn's API is organized around three main interfaces: *program*, *object*, and *helpers*. The `drgn.Program` object represents the ELF currently being debugged. As the top-level interface, it provides methods to look up type definitions (e.g., struct members) and read from process memory. `drgn.Object` consist of variables, constants, functions (all memory references to their corresponding program data) and computed values. Finally, the `drgn.helper` package encapsulates common debugging tasks such as traversing linked lists or working with virtual addresses and page frames. Analysts use drgn either in scripted mode, or interactive. Both have the same syntax and API, the only difference is whether the library is used in the analyst's scripts, or at the Python interpreter CLI.

Use Cases and Limitations

A typical use case is converting the program's values into Python values that can be referenced as they are in the source of the program under inspection. For example:



```
>>> print(repr(prog['init_task'].nsproxy.mnt_ns.mounts + 1))
Object(prog, 'unsigned int', value=34)
>>> prog['init_task'].nsproxy.mnt_ns.pending_mounts > 0
False
```

Figure 9: Accessing file system mounts in the current Linux namespace [61]

The `drgn.helpers` package assists when working with specific program types. Currently only Linux kernel helpers are provided, but sub-packages could be created for other types (e.g., libc). An example of using helpers to create a simple “ps”:

```
1 from drgn.helpers.linux.pid import for_each_task
2 print('PID      COMM')
3 for task in for_each_task(prog):
4     pid = task.pid.value_()
5     comm = task.comm.string_().decode()
6     print(f'{pid:<10} {comm}')
```

Figure 10: Printing process ids and their associated executable tasks [61]

`drgn` is useful when DWARF debugging symbols are available for a program under inspection [62]. This information is used to find variables and other symbols and manipulate them. Without the DWARF symbols, `drgn`’s main feature, intuitive program analysis via Python, cannot work effectively. Memory can still be read and mutated, but it requires additional logic, which negates the major benefit of using the library.

`drgn` relies on `libdwfl` from the Open Source `elfutils` project to parse debugging symbols and make them accessible via Python.

4.3 Dynamic Binary Instrumentation

4.3.1 Qiling

Reference Link	https://www.qiling.io
Target Type	Binary – ELF
Host/Target Operating System	Linux; BSD; macOS; Windows
Host/Target Architecture	x86 (32, 64); ARM (32, 64); MIPS (32, 64); Spark; M68K
Initial Release	November 2019
License Type	Open-Source
Maintenance	Maintained by Lau Kai Jern, Dr. Nguyen Anh Quynh, Ding Tian Ze, Sun Bo Wen, Chen Hui Tao, and Yu Tong

Overview

Qiling is a cross-platform tool that can perform emulation, binary instrumentation, disassembly, sandboxing, and any combination of these techniques, on raw binaries. While it can run on any raw binary, Qiling can also identify binaries of specific executable formats which allows it to recognize OS level concepts like dynamic libraries and I/O handling. When executing a binary,



Qiling can perform cross-platform and cross-architectural analysis. Qiling also provides a python framework to build dynamic analysis tool to enable features like hot patching and dynamic hooking. [63] [7]

Design and Implementation

Qiling is built on top of Unicorn Engine and QEMU as a single python-based framework that encapsulates all the functionality of a binary emulator. These functions include process redirection, hot patching, and code injection. Qiling can perform partial execution, as well as patch unpacked binary content. Qiling can run cross-platform emulation between Linux, macOS, Windows, and/or BSD. For example, a target binary recognized as a MachO file, can be run on top of Windows. It can recognize PE, MachO, and ELF file headers and be context-aware of system level calls specific to that OS. Like Unicorn, Qiling is also cross-architectural and supports ARM/ARM64, M68K, MIPS, Sparc, x86/x64 architectures. The framework provides plugins for disassemblers like Ghidra and IDAPro and can be used to build other tools in Python.

Qiling can be imported into a python file and hook into an executable by specifying the path in a new Qiling object. From there, Qiling can run the binary on the specified path and add additional library calls to interact with the process during run-time. These library calls include *patch* to apply hotpatching, *stack_push* to set up stack memory, various calls to move stack pointers, *hook_address* to hook at a specified address, etc.

Use Cases and Limitations

Unlike Unicorn, Qiling can understand OS level calls when it recognizes MachO, PE, or ELF files. Qiling can best be used to run and analyze raw binaries or specific executables from various sources due to its flexibility in being able to run from different architectures as well as different operating systems. As a tool, Qiling would be useful in running potential malware binaries, such as in IoT devices, through its ability to hook, examine, and patch binaries to execute certain branches.

4.4 Dynamic and Symbolic Binary Analysis

4.4.1 Triton

Reference Link	https://triton.quarkslab.com/
Target Type	Binary (Application)
Host/Target Operating System	Linux; macOS; Windows;
Host/Target Architecture	x86 (32, 64)
Initial Release	2015
License Type	Open-Source
Maintenance	Maintained by Quarkslab



Overview

Triton is a dynamic binary analysis tool which is intended primarily as a concolic execution framework. It is based on Intel Pin [52] and, among other features, encompasses a taint engine, symbolic execution engine, an interface for SMT solving. These can be used to perform trace analysis, symbolic execution, hybrid execution, constraint solving, etc. It allows the user to gather code coverage information, modify registers and memory during run-time, and replay traces. This is a well-maintained and well-documented tool that can be used via its Python bindings. [49], [64], [65], [66], [67]

Design and Implementation

Triton is used primarily with Python bindings. It is built on top of Intel Pin, which is used to implement its capabilities, but additionally a user can use some of Pin’s internal functions via the Python bindings, such as adding callbacks. A diagram of Triton’s architecture is shown in Figure 11 [65].

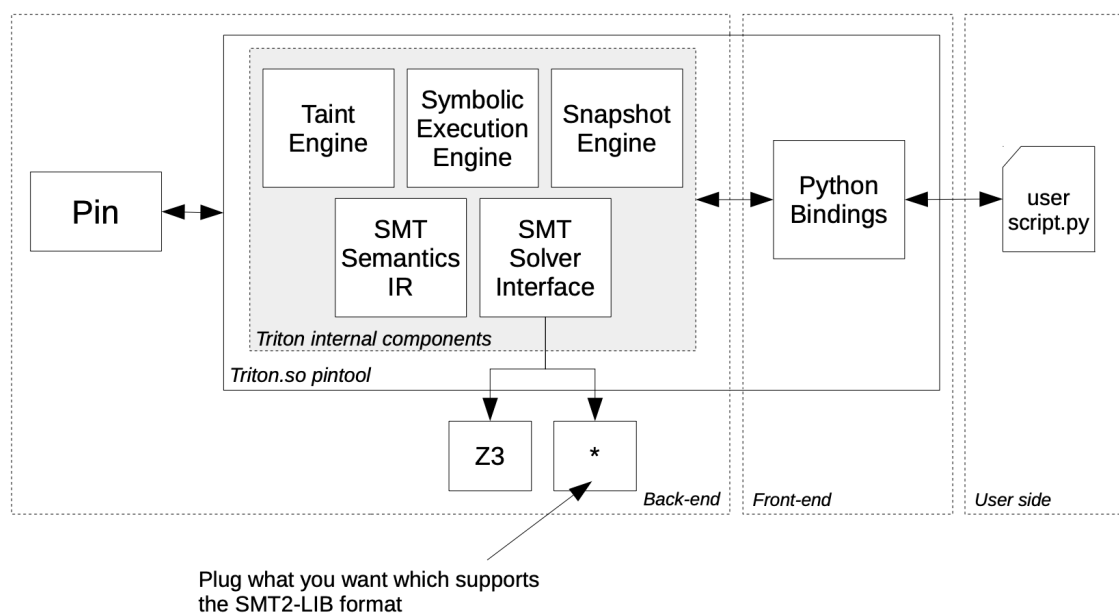


Figure 11: Architecture of Triton [65]

Taint Engine: Triton’s taint engine, which allows the user to perform dynamic taint analysis, is one of its most well-regarded features. Intel Pin is a commonly used tool for dynamic taint analysis, and so by providing a dynamic taint analysis framework built around Intel Pin, Triton has made it easy to leverage Intel Pin’s capabilities in this area.

Triton’s main author, Jonathan Salwan, describes the design of the taint engine in [65],

“This engine applies an over-proximation but that doesn't affect the precision. In exploit development, what the user wants in reality is knowing if the register is controllable by



himself and know what values can hold this register. Answering to this question only with the taint analysis is pretty hard because a lot of instructions have an influence on the value that can hold a register. (Path conditions, arithmetic operations...)

“So, our personal reflection about this is: **How can we gain time without losing precision?**

“Symbolic execution offers us the possibility to answer at the question "what value can hold a register?", but applying a symbolic execution and asking a model at each program point if a register is controllable is pretty expensive. Therefore, we use an over-approximation to fix the loss of time and if a register is tainted, we ask a model for the precision [65].”

Symbolic Execution: Triton’s underlying methodology is described in a comparative study of symbolic execution tools, *On Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs* [59], as follows,

“Triton is a dynamic symbolic execution tool based on binaries. It automatically accepts symbolic variables from the standard input. During symbolic execution, Triton firstly runs the programs with concrete values and leverages Intel PinTool to trace related instructions, then it lifts the traced instructions into the SSA (single static assignment) form and performs symbolic analysis. If there are alternative paths found in the trace, Triton generates new test cases via symbolic reasoning and employs them as the concrete values in the following rounds of concrete execution. This symbolic execution process goes on until no alternative path can be found [59, p. 10].”

Use Cases and Limitations

Triton provides a certain ease of use as a reasonably well maintained and documented tool with an easy to use Python API. This API even facilitates using Pin itself by allowing certain Pin functions to be called via these Python bindings; one can also use it to interface with Z3 or a constraint solver of their choice. In this way Triton can be used as a wrapper around other tools.

Triton’s most developed feature is its taint engine, which makes taint tracking its best use case. However, there are many reasons one would choose to use Triton for concolic execution, including using it for the Pin bindings.



4.5 Constraint Solvers

4.5.1 Just Fuzz It Solver

Reference Link	https://github.com/mc-imperial/jfs
Target Type	Floating Point SMT
Host Operating System	Linux; macOS
Host Architecture	x86 (32, 64)
Initial Release	July 2019
License Type	Open-Source
Maintenance	Last Commit August 2019

Overview

Just Fuzz It Solver (JFS) is a constraint solver for floating-points which implements a novel approach proposed in the paper, *Just Fuzz It: Solving Floating-Point Constraints using Coverage-Guided Fuzzing* [50]. Satisfiability Modulo Theories (SMT) solvers are particularly challenged by floating-point constraints, which is discussed in Section 3.1. Rather than using a typical bit-blasting approach like the one used in Z3 [68], JFS models the constraints as a program and then fuzzes it to find a satisfiable solution. However, unlike Z3 this method does not prove non-satisfiability. The paper presents evidence that JFS competes with state-of-the-art floating-point SMT solvers and outperforms them when combined with other SMT solving approaches (i.e., it complements the existing state-of-the-art). [50], [69]

Design and Implementation

Given a set of floating-point constraints as an SMT, JFS first models them as a program. It does so by first taking as input the values which correspond to the variables in the original SMT. It then uses `if` statements to create a branch for each constraint, in which the branch condition is the same as that constraint. An input to this program which causes only true branches to be taken would satisfy the original SMT. Conversely, any input which takes a false branch immediately causes the program to return, thus not wasting time on a non-satisfying input. However, there is no way to prove non-satisfiability given this method. A timeout is set for the fuzzer if no satisfying input is found quickly enough. Figure 12 [50] shows an example of this an original SMT, and then resulting program.



Listing 1: An example conjunction of floating-point constraints in the SMT-LIBv2.5 format.

```
1 (declare-fun a () Float64)
2 (declare-fun b () Float64)
3 (define-fun div_rne () Float64 (fp.div RNE a b))
4 (define-fun div_rtp () Float64 (fp.div RTP a b))
5 (assert (not (fp.isNaN a)))
6 (assert (not (fp.isNaN b)))
7 (assert (not (fp.isNaN div_rne)))
8 (assert (not (fp.isNaN div_rtp)))
9 (assert (not (fp.eq div_rne div_rtp)))
10 (check-sat)
```

Listing 2: A translation of the constraints in Listing 1 to a C++ program.

```
1 int FuzzOneInput(const uint8_t* data, size_t size) {
2     double a = makeFloatFrom(data, size, 0, 63);
3     double b = makeFloatFrom(data, size, 64, 127);
4     if (!isnan(a)) {} else return 0;
5     if (!isnan(b)) {} else return 0;
6     double a_b_rne = div_rne(a, b);
7     double a_b_rtp = div_rtp(a, b);
8     if (!isnan(a_b_rne)) {} else return 0;
9     if (!isnan(a_b_rtp)) {} else return 0;
10    if (a_b_rne != a_b_rtp) {} else return 0;
11    return 1; // TARGET REACHED
12 }
```

Figure 12: Example given in [50, p. 3]

Once the program is constructed, it is fuzzed for an input which reaches the target (i.e., the point reached when every true branch is taken). JFS uses LibFuzzer [70] to do this and offers two options for seed generation. In *naive seeds* mode, LibFuzzer is seeded with a buffer of all zeros and a buffer of all ones. In *smart seeds* mode, LibFuzzer is given a variety of seeds which correspond with the different types of floating-points. The latter mode is the one used to evaluate JFS against other constraint solvers.

The JFS implementation uses Z3 to parse constraints and generates the program in C++, which is then compiled with Clang [16] using the LibFuzzer flags.

Use Cases and Limitations

There are currently several approaches SMT solvers use for floating-points, which are described in [60] as follows:

- Bit-Blasting converts the floating-point SMT to a boolean SAT problem. This approach is used by many well-known SMT solvers including Z3. One issue with this approach is that these SAT problems can grow quickly, reducing scalability. The paper referenced above implements a new library of encodings to help reduce floating-points to bit-vectors that can be used in solvers. [60] presents evidence that use of this tool showed significant performance gain in relation to the current state of the art.



- Interval arithmetic techniques are another set of approaches which rely on the numerical methods techniques developed to handle issues in floating-point arithmetic like rounding errors. Both MathSAT and Z3 implement this into their approach, and the SMT solver COLIBRI uses this as a complete method.
- Some SMT solvers convert the problem into a mathematical optimization problem. These include ZSat and goSAT. JFS's technique is similar to this approach.
- Other approaches use heuristics to reason about floating-points, these include CORAL and FloPSy

The authors found that JFS performed competitively with state-of-the-art constraint solvers on floating-point constraints, and that when combined with solvers which used different techniques, such as MathSAT [71] (which uses bit-blasting), they were able to outperform Z3 [50]. This implies that JFS is useful as a complement to the current state of the art. Using it in this way would also mitigate a major limitation of the JFS approach, which is that it cannot prove non-satisfiability (Z3, for instance, can do so). Another limitation, when comparing JFS to Z3, is usability. Z3 is an actively maintained tool, while JFS is an experimental solver that is not actively maintained.

4.6 Trends

Increased Usability and Capability

Many of the techniques discussed in this section have been around for over a decade or more, but over the last few years many tools have been released which make it far easier for the user to leverage them. These tools include ones discussed in this report, and its previous version. Frida and Qiling make it very simple to hook into a binary and manipulate it. BAP 2.0 now offers new functionality designed to further enable program analysis techniques. Tools like angr, Miasm and Triton have easy to use symbolic execution engines that save users the trouble of constructing their own, and offer a wide array of functionality. These are driven by increasingly effective constraint solvers, which in turn makes symbolic execution more feasible. Debuggers are also more sophisticated, notably rr and Windbg Preview both allow users to execute backwards. Even though the underlying technique is far from novel, the tools which implement it are increasingly usable and capable.

Frida

Frida is a binary instrumentation tool which was discussed in the previous version of this report. Over the last few years it has become an increasingly popular tool for hooking into applications, like messaging programs. The Google Project Zero researcher, Samuel Groß recently released a set of tools for instrumenting the iOS messaging platform iMessage with Frida. Frida is also the instrumentation tool behind the fuzzer Frizzer (Section 5.2.2). Frida is notable because it has become a staple in certain areas of vulnerability research, like mobile applications, in a way that most tools are not.



5 Fuzzing

5.1 Technical Overview

Fuzzing is a form of software testing which feeds semi-random inputs into a computer program to trigger a crash or other unintended behavior. While fuzzing could be considered just another form of dynamic analysis, the plethora of available fuzzers and fuzzing techniques warrants a category all its own. Although simple in concept, an entire field of research has formed around fuzzing and its various forms to find more effective and faster methods to mutate or generate input.

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

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

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

A black-box fuzzer is one which treats its target like a ‘black-box’ and lacks any knowledge of program internals. These fuzzers are easy to construct, but are not the most effective at bug-finding. The most common type of fuzzer discussed in this report is that of the grey-box variety, which is driven by some kind of instrumentation, whether it is compiled in from source code, or instrumented as a binary. A white-box approach is one which leverages additional knowledge of the program, from its source code and by using program analysis techniques, such as combining symbolic execution and constraint solving with fuzzing.

Mutation-based Fuzzing

Mutation-based fuzzing does not require input to be formatted based on some predetermined grammar, rather this incredibly common type of fuzzing takes example inputs in the form of seeds, which it then mutates to create input for the program. Although mutation can be done blindly in a black-box scenario, mutation-based fuzzing is typically grey-box: the program binary is instrumented to glean coverage and other information, which the fuzzer then uses to mutate the seeds, often to find new or specific paths. This can also be done as white-box fuzzing, with mutation aided by program analysis. One incredibly well-known mutation-based fuzzer is AFL [72].



Several mutation-driven fuzzers are included in this section, including honggfuzz [73] and Frizzer [74].

Generation-based Fuzzing

Generation-based fuzzing takes some kind of generator, often in the form of a grammar, and uses that to generate input into a fuzzer. Generation-based fuzzers are useful for fuzzing programs with complex input formats. In the case of grammar-based fuzzers, a grammar is designed to model syntactically correct inputs, and then used to randomly generate them. But generation-based fuzzers are not simply limited to grammar-based fuzzers. In the previous version of this report, two fuzzers targeting Javascript Just-In-Time (JIT) Engines were highlighted that adopted different techniques towards generating semantically and syntactically correct Javascript inputs [28], [75]. One of these, the fuzzer Fuzzilli [28], was built on a custom IR, FuzzIL, that was initially generated, and then lifted to Javascript.

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

As discussed further in Trends, grammar-based fuzzers can be incredibly slow, which limits their effectiveness. One recent fuzzer discussed in this section which attempts to address this is the F1 Fuzzer.

Hybrid Fuzzing

Hybrid fuzzing leverages the benefits of fuzzing (less time and memory complexity) with those of symbolic execution (knowledge of the program, such as path constraints). This is a form of white-box fuzzing, in which the fuzzer is assisted by symbolic execution. This method has been implemented in a variety of fuzzers, such as SAGE [76], and many of those which competed in the DARPA Cyber Grand Challenge (CGC) [77]. However, hybrid fuzzing often suffers from the same performance issues as symbolic execution. A recent, and arguably successful, implementation of the technique can be seen in the QSYM fuzzer [58], which was discussed in the previous version of this report.

Measuring Fuzzer Performance

How to best measure the efficacy of a fuzzer has recently been a frequent discussion among fuzzing researchers, especially regarding the mutation-based variety. The ability to effectively determine the performance of a fuzzer relative to others has been limited by the variance and lack of consensus in how researchers measure their fuzzers. Many fuzzing papers will test on different software, for different amounts of time, and count different kinds of results as a success. The 2018 paper, *Evaluating Fuzz Testing* [78], examined how a spate of recent fuzzing research



was evaluated, and recommended certain testing standards. Many subsequent fuzzing papers have followed this paper’s advice, including tools discussed in this report, such as EnFuzz and Retrowrite. However, more progress is necessary to determine how best to measure the performance of fuzzers, and there is far from a consensus in the community.

5.2 Mutation-based Fuzzing

5.2.1 honggfuzz

Reference Link	https://honggfuzz.com
Target Type	Binaries (Linux), Source (all supported Oss)
Host Operating System	GNU/Linux, FreeBSD, NetBSD, macOS, Windows/Cygwin, Android
Target Operating System	GNU/Linux, FreeBSD, NetBSD, macOS, Windows/Cygwin, Android
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	Version 0.6: August 2015
License Type	Open-Source
Maintenance	Maintained by Google

Overview

honggfuzz is a multi-process, multi-threaded fuzzer that adaptively modifies an input corpus based on process signals and code coverage instrumentation [79] [80] [73]. Developed by Google, this is a mutation-based fuzzer designed to be fast, automatically parallelized and run for an extended amount of time. honggfuzz has been responsible for a significant number of bugs found in major software projects, including OpenSSH, OpenSSL, and common interpreters such as Python, Ruby and PHP [80].

Design and Implementation

honggfuzz runs all files in the input corpus through the target application and identifies which files result in increased code coverage (depending on coverage instrumentation fidelity). Those that result in new or increased coverage are added to an in-memory dynamic input corpus. honggfuzz then selects a random input from the dynamic corpus to mutate. If the new input extends coverage it is added to the dynamic input corpus to be used in future fuzzing rounds. This feature can also be used to generate a minimized corpus.

honggfuzz can also run in ‘persistent fuzzing’ mode to increase the speed of testing APIs within the same process. To use persistent fuzzing, the test binary must be compiled with the honggfuzz-specific clang wrapper hfuzz-clang and provide the test function to fuzz, as shown in Figure 13.



```

1  // fuzz_target.cc
2  extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
3      DoSomethingInterestingWithMyAPI(Data, Size)
4      return 0;
5  }

```

Figure 13: honggfuzz clang wrapper example. [79]

Alternatively, the fuzzed program can be modified to call the HF_ITER honggfuzz API function that will load the test input into memory and then start fuzzing, as shown in Figure 14.

```

1  extern HF_ITER(uint8_t** buf, size_t* len);
2
3  int main(void) {
4      for (;;) {
5          size_t len;
6          uint8_t *buf;
7
8          HF_ITER(&buf, &len);
9
10         ApiToBeFuzzed(buf, len);
11     }
12 }

```

Figure 14: Example of honggfuzz HF_ITER API. [79] [80] [73]

honggfuzz can be extended by using external mutators to generate new inputs rather than the built-in honggfuzz mutators. A postprocessor can also be specified to run over the test inputs generated via the mutations, which can be useful in fixing up checksums or other known gatekeeper fields.

honggfuzz can also perform non-persistent fuzzing with or without instrumentation, QEMU-mode (black-box instrumentation), and corpus minimization. However, despite this array of features, using this fuzzer requires a POSIX compliant operating system, which excludes Windows. For hardware-based code coverage a GNU/Linux kernel v4.2 or greater is also required.

Use Cases and Limitations

In addition to a standalone fuzzer, honggfuzz has also been integrated into multiple projects and extended by others. Its most notable feature is its persistent mode, which allows it to be leveraged for long periods of fuzzing. Combined with the focus on speed, honggfuzz is an attractive option for serious, long-term bug finding efforts.

However, these abilities can only be used with POSIX compliant operating systems, and further, the hardware-based code coverage can only apply to a specific GNU/Linux kernel. These requirements limit honggfuzz's reach and applicability.



5.2.2 Frizzer

Reference Link	https://github.com/demantz/frizzer
Target Type	Binary
Host Operating System	Linux Undocumented, but may work on: macOS, Windows, iOS, Android
Target Operating System	Linux; Windows
Host Architecture	x86 (32, 64) Undocumented, but may work on: ARM (32, 64)
Target Architecture	x86 (32, 64) Undocumented, but may work on: ARM (32, 64)
Initial Release	October 2019
License Type	Open-Source
Maintenance	Last commit October 2019

Overview

Frizzer [74] is a coverage-guided fuzzing engine built using the dynamic code instrumentation toolkit Frida [8] and the fuzzer Radamsa [81].

Design and Implementation

Users create a Frizzer configuration file for their target that specifies the name of the process to attach to and the address of the function to hook. To run a Frizzer job, an INI-format configuration file must be created for the process being fuzzed. An example INI file is shown in Figure 15 [74].

```
frizzer init tmpprojdir
cat > tmpprojdir/config <<EOF
[fuzzer]
log_level      = 3 # debug
debug_mode     = false
[target]
process_name   = "test"
function       = 0x401256
host           = "localhost"
port           = 7777
remote_frida   = false
fuzz_in_process = false
modules = [
    "/home/dennis/tools/frida-fuzzer/tests/simple_binary/test",
]
EOF
```

Figure 15: An example Frizzer INI file [74, p. "frizzer/tests/simple_binary/run_tests.sh"]

Fuzzing can be performed either with a TCP connection to the host and port specified by the keys of the same name, or via injection into the process at a specific address. The input method is determined by the setting of the Boolean key *fuzz_in_process*.



After choosing a fuzzing method, the *process_name* key should be modified to match the process that will be injected. The *function* key is the address of the function in the binary to be analyzed and fuzzed (for in process fuzzing). The *modules* key lists modules that will be analyzed for code coverage. Limiting the modules list to just the main binary and excluding standard libraries should lower runtime and provide more useful results.

Run the following to set up a frizzer project, here named “projdir”:

```
frizzer init projdir
```

Next, a corpus directory of sample inputs must be generated. Create the corpus directory, indir, with a file for each sample input to be used for the fuzzing. Add the corpus to the project using

```
frizzer add -p projdir indir
```

Launch the program under test. Then launch the fuzzing job using

```
frizzer fuzz -p projdir
```

Finally, analyze the output text to see test cases that uncovered new code coverage results and test cases that resulted in crashes. The corpus of inputs can then be modified based on inputs that produced interesting coverage results or crashed the program under test.

Use Cases and Limitations

Frizzer currently requires some minor configuration modifications to enable fuzzing against a new target process. Documentation is lacking, so the correct configuration changes and other fixes must be determined from examining two similar examples bundled with the application, along with Python stack backtraces and Frizzer source code examination. There has been no activity on the repository since the initial commit in October 2019.

The most critical limitation of Frizzer is that it only fuzzes the first argument of a function and assumes that argument is a C-style string (i.e., `char*`). This not only severely limits capabilities against C programs, but it also prevents C++ programs from being effectively analyzed with Frizzer since C++ member functions normally have the object as the first argument. Attempts to test fuzzing a function with multiple arguments resulted in arguments after the first being modified to contain values different than those originally passed to the function by the program under test, further verifying the limitation of fuzzing single argument functions.

One limitation identified in [82], is that the Frizzer configuration file does not support use of function names for the function option. Instead, the user must determine the address of the function itself, even though Frida supports symbol lookups. This can be overcome using any number of tools. [82] uses the reverse engineering tool, radare2, to get the address of a function given its symbol.



It is also important to keep in mind that finding a vulnerability in an internal function does not mean that the program containing the function is vulnerable. A path through the program with the payload data must still be determined.

Despite Frizzer's limitations, it is a very new public project that has a lot of potential and a lot of room for growth. The ability to fuzz an individual function by simply creating a short configuration file and a corpus of sample input is very powerful and can greatly speed up the time it takes to setup targeted fuzzing. [74]

5.2.3 Redqueen

Reference Link	https://github.com/RUB-SysSec/redqueen
Target Type	Application and Kernel Binaries
Host Operating System	Linux
Target Operating System	Linux; macOS; Windows
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	September 2019
License Type	AGPLv3
Maintenance	Last commit September 2019

Overview

Redqueen is a general purpose fuzzer [83] built on top of the kAFL kernel fuzzer [84]. Redqueen's main contribution is the ability to efficiently and automatically fixup gatekeeper values in a test input, such as magic numbers and checksums. Redqueen can automatically fixup these values without requiring source code or relying on costly taint tracking or symbolic execution. This is accomplished by analyzing the program state and correlating it with the input test case. [83], [85], [86]

Design and Implementation

Redqueen is built on the kAFL kernel fuzzer, and obtains coverage information via Intel Processor Trace (IPT), which is then used to prioritize new inputs. Redqueen has extended kAFL to support the fuzzing of userland applications within a VM, and to include the Radamsa mutator [81], in addition to the standard AFL mutations.

Redqueen's main enhancement over kAFL is its ability to automatically fixup magic numbers and checksums. Redqueen's fixup algorithm works on the assumption that any magic bytes checked during the execution of program will come from the input. The fixup process is accomplished using Virtual Machine Introspection (VMI) to place breakpoints on comparison instructions, such as *cmp*, and functions such as *memcmp*, *strcmp*, etc. Redqueen uses these breakpoints to extract the values used during each comparison. The values are then encoded in several different ways (e.g., converting the byte values to ASCII digits, reversing the bytes, etc). Finally, Redqueen searches the input test case for the encoded bytes and replaces them with the



values that they are compared against. Thus, when the program reaches the comparison again, it will take the opposite path, as the replaced values will now be used.

Use Cases and Limitations

Redqueen extends kAFL to support both userland and kernel fuzzing. Several other enhancements to kAFL have been made, such as adding Windows fuzzing agents and many bug fixes. As such, Redqueen appears to be a much more mature fuzzer than the original kAFL. However, as Redqueen extends kAFL, it shares some of the same limitations [83]:

1. Redqueen requires a processor that supports IPT. IPT with address filtering was first available in Intel's Skylake architecture, and as such, Redqueen requires a relatively recent Intel processor. Additionally, IPT is only supported on a subset of new processors and Intel does not provide a list of processors which support IPT. Therefore, acquiring a processor specifically for use with IPT is non-trivial.
2. Redqueen is not actively maintained. However, unlike kAFL, Redqueen was released much more recently.
3. The kAFL/Redqueen fuzzing algorithm is better suited towards fuzzing blobs of data than the more traditional kernel attack surfaces, such as the syscall and ioctl handlers. However, unlike kAFL, Redqueen also supports userland applications, which adds an additional use case.

5.3 Generation-based Fuzzing

5.3.1 F1 Fuzzer

Reference Link	pypi.org/project/F1-Fuzzer/
Target Type	Source; Binary
Host Operating System	Linux; macOS; Windows
Target Operating System	Linux; macOS; Windows
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	November 2019
License Type	Open-Source
Maintenance	Maintained by Andreas Zeller

Overview

F1 Fuzzer is a grammar-based fuzzer, designed to generate inputs at high speed, introduced in *Building Fast Fuzzers* by Rahul Gopinath and Andreas Zeller [87]. Past grammar-based fuzzers have been hampered by poor performance. As their paper notes, even the fastest grammar-based fuzzer (before this) produced inputs “a thousand times slower” than the fastest random fuzzer



[87, p. 1]. This is achieved through a variety of means including precomputing key expansions, as well as implementing and compiling the grammar.

The F1 Fuzzer is written in Python, and is available through the pip installer [88]. The fuzzing researcher Brandon Falk subsequently released an implementation of their research in Rust, called FZero, which generates inputs even faster than F1 [89].

Design and Implementation

An example grammar and a Python implementation of that grammar, which are shown in Figure 16 and Figure 17, respectively [87]. F1 takes as user input a grammar in Python form, and uses that to generate fuzzing inputs.

```

$$\begin{aligned} \langle START \rangle &::= \langle expr \rangle \\ \langle expr \rangle &::= \langle term \rangle \\ &\quad | \langle term \rangle '+' \langle expr \rangle \\ &\quad | \langle term \rangle '-' \langle expr \rangle \\ \langle term \rangle &::= \langle factor \rangle \\ &\quad | \langle factor \rangle '*' \langle term \rangle \\ &\quad | \langle factor \rangle '/' \langle term \rangle \\ \langle factor \rangle &::= \langle integer \rangle \\ &\quad | \langle integer \rangle '.' \langle integer \rangle \\ &\quad | '+' \langle factor \rangle \\ &\quad | '-' \langle factor \rangle \\ &\quad | '(' \langle expr \rangle ')' \\ \langle integer \rangle &::= \langle digit \rangle \langle integer \rangle \\ &\quad | \langle digit \rangle \\ \langle digit \rangle &::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' \end{aligned}$$

```

Figure 16: An example grammar from [87, p. 1]



```

1  EXPR_GRAMMAR = {
2      "<start>":
3      | ["<expr>"],
4
5      "<expr>":
6      | ["<term> + <expr>", "<term> - <expr>", "<term>"],
7
8      "<term>":
9      | ["<factor> * <term>", "<factor> / <term>", "<factor>"],
10
11     "<factor>":
12     | ["+<factor>",
13        "-<factor>",
14        "<expr>",
15        "<integer>.<integer>",
16        "<integer>"],
17
18     "<integer>":
19     | ["<digit><integer>", "<digit>"],
20
21     "<digit>":
22     | ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
23 }
24
25
26 def gen_key(grammar, key)
27     if key not in grammar: return key
28     return gen_alt(grammar, random.choice(grammar[key]))
29
30
31 def gen_alt(grammar, alt):
32     return ''.join(gen_key(grammar, t)) for t in alt
33
34 gen_key(expr_grammar, '<start>')

```

Figure 17: A Python implementation of the grammar presented in Figure 16 from [87, p. 2]

F1 uses a variety of tactics to produce a fast and space efficient fuzzer. These range from precomputing key expansions, to implementing and compiling the grammar. They also tackle the issue of stack memory usage. The result is an extremely fast grammar based fuzzer that, according to their tests, outperforms the current state of the art by several orders of magnitude. [87]

Grammar Compilation – First, the F1 fuzzer compiles a grammar by translating each nonterminal¹ to a function in a fast, low level programming language like C, and then compiling it to machine code. F1 then performs ‘supercompilation’ by using symbolic analysis to search for code that interacts with input and eliminates as much redundancy as possible. This produces a generator which executes at significantly increased speed.

Space Complexity – As discussed in Section 5.1, memory usage is an issue with grammar-based fuzzers. When grammars are highly recursive, generating input to satisfy them can lead to stack exhaustion. F1 addresses this by defining both a minimum depth of expansion and a stack

¹ A nonterminal is any symbol in the grammar which can be expanded into another symbol. For instance, in Figure 16, <integer> could be expanded into a <digit> or a sequence of them, <digit>...<digit>. <digit> is a terminal symbol and cannot be expanded to be any other symbol, rather it can only be one of 10 numbers.



budget. Once the budget is surpassed, it only uses completely expanded strings which meet the minimum depth.

FZero – After Gopinath and Zeller’s paper was published, it was implemented in the Rust programming language by Brandon Falk. His tests show FZero to be even faster than F1 [89].

Usage – To use F1 (or FZero), one must first manually define a grammar as a Python implementation, like that shown in Figure 17. Afterwards, F1 compiles that grammar into an executable which the user runs to generate inputs.

Use Cases and Limitations

The obvious use case for the F1 Fuzzer is as a grammar-based fuzzer, but it can also be used to test platforms which require high throughput [87, p. 2]. Due to the significant speed-up offered by F1, grammar-based fuzzing can become a far more fruitful endeavor. Increased speed results in more inputs being tested, which can potentially yield more crashes, or other unintended behavior.

Grammar-based fuzzing is however limited by the requirement that the user construct a grammar. However, there are emerging options that may ease this constraint: “Recently, however, a number of approaches have been proposed to infer both regular languages as well as context-free grammars either by grammar induction from samples or by dynamic grammar inference from program code [87, p. 2].”

5.4 Platform Specific Fuzzing

5.4.1 kAFL

Reference Link	https://github.com/RUB-SysSec/kAFL
Target Type	OS kernels without source code
Host Operating System	Linux
Target Operating System	Linux; macOS; Windows
Host Architecture	x86 (64)
Target Architecture	X86 (64)
Initial Release	2017
License Type	GNU-GPL v2
Maintenance	Last commit November 2017

Overview

kAFL is a coverage-guided kernel fuzzer developed by researchers at the Ruhr University Bochum, who described their work in *kafl: Hardware-assisted feedback fuzzing for OS kernels* [84]. Unlike previous coverage-based kernel fuzzers, which use QEMU [90] or compile-time instrumentation [91], kAFL uses Intel Processor Trace (IPT) to quickly obtain coverage



information from kernel execution without requiring source code. While kAFL is much faster than other blackbox approaches, its use of IPT reduces the number of computers which can run kAFL, as not all processors support IPT. While kAFL has not been officially maintained, several forks [92], [93] on GitHub have spawned since its initial release in 2017 that update kAFL to work with the latest QEMU and Linux kernel. [84]

Design and Implementation

kAFL is built on top of QEMU/KVM, with the target Operating System (OS) running inside of a Virtual Machine (VM). A small agent runs within the target VM and is used to gather information about the OS and execute new testcases. kAFL uses a modified KVM driver to selectively enable IPT only when the guest is executing, and configures an address and CPU mode filter to ensure only the VM's kernel coverage is tracked. kAFL then starts the test input inside the VM, gathers the IPT trace data, parses it using a custom IPT decoder and the Capstone disassembly framework [5], and generates an AFL-style bitmap of basic block transitions. Like AFL, the bitmap is then used to select the next test input. While the original AFL fuzzer is single threaded, kAFL manages multiple parallel processes. As a large amount of kAFL's work is not CPU-bound, using several parallel processes greatly improves performance.

Use Cases and Limitations

kAFL is designed for kernel fuzzing with particular emphasis on speed and the ability to fuzz an OS's kernel without access to the source code. While kAFL's novel approach of utilizing IPT for kernel tracing can greatly increase the performance of kernel fuzzing, kAFL suffers several limitations which have prevented it from gaining widespread adoption [84]:

1. kAFL requires a processor that supports IPT. IPT with address filtering was first available in Intel's Skylake architecture, and as such, kAFL requires a relatively recent Intel processor. Additionally, IPT is only supported on a subset of new processors and Intel does not provide a list of processors which support IPT. Therefore, acquiring a processor specifically for use with IPT is non-trivial.
2. kAFL lacks consistent maintenance and does not have an active user community. While a few users (e.g., [92] and [93]) have made minor updates to kAFL, it has largely been left untouched. Additionally, despite the framework supporting macOS and Windows, kAFL does not provide any macOS or Windows agents, other than simple proof-of-concept fuzzers that are not useful for analyzing the macOS and Windows kernel.
3. The kAFL fuzzing algorithm is better suited towards fuzzing blobs of data, rather than fuzzing a kernel. Therefore, developing a fuzzing agent which fuzzes more traditional kernel attack surfaces, such as the syscall and ioctl handlers, would require substantial effort.

Despite these disadvantages, kAFL is a promising technology that a dedicated researcher can apply to efficiently analyze an OS's kernel without requiring source code.



5.5 Frameworks

5.5.1 Enfuzz

Reference Link	https://github.com/enfuzz/enfuzz
Target Type	Source
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	2019
License Type	Open-Source
Maintenance	Last commit 05/03/2019

Overview

EnFuzz is a fuzzing framework that uses an ensemble of multiple, diverse fuzzers, and was presented in *EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers* [94]. By combining multiple fuzzers, EnFuzz attempts to achieve better performance and results than any single fuzzer. The set of fuzzers used in EnFuzz are AFL [72], libFuzzer [70], AFLFast [95], Fairfuzz [96], and Radamsa [81]. Combining these fuzzers enable EnFuzz to leverage three (3) different heuristics: coverage information granularity, input generation strategy, and seed mutation selection. EnFuzz is scalable and can integrate other fuzzers as desired. [94], [97]

Design and Implementation

EnFuzz is based on the ensemble fuzzing architecture illustrated in Figure 18:



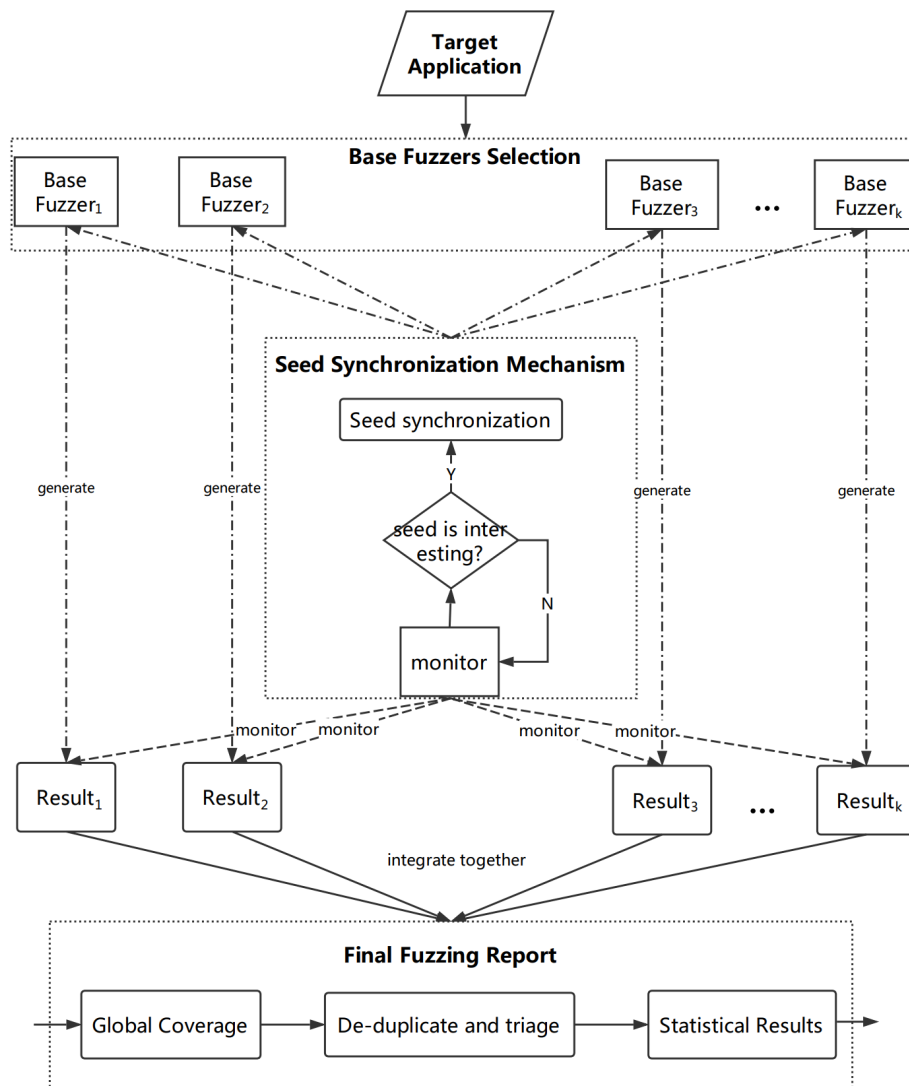


Figure 18: Ensemble Fuzzing Architecture [94, p. 4]

For each fuzzer EnFuzz creates a base fuzzer object and feeds it a seed based on its seed synchronization mechanism. It continuously monitors the seed and is able to provide additional seeds to all the base fuzzers in a timely manner. Once the results are available, they are post-processed, and provided to the user.

EnFuzz is controlled by a graphical user interface (GUI), an example of which is provided in Figure 19:

Figure 19: EnFuzz UI step to upload source code [97]

The UI is used to create a fuzzing job following a few basic steps:

1. Select target application OS
2. Upload the source code (accepts tar.gz and zip files)
3. Upload the initial seed (accepts tar.gz and zip files)
4. Select the set of base fuzzers to use
5. Confirm the test selections

Use Cases and Limitations

Ensemble fuzzing works best with a diverse set of fuzzers. The five fuzzers that comprise EnFuzz seem to provide enough diversity because they combine three heuristics metrics. Test results generated while fuzzing real-world applications show an overall increase in performance in all the categories tested when compared to running each fuzzer in isolation. For example, EnFuzz demonstrated a 76% improvement on the average number of unique bugs found in comparison to AFL. In the same test, the next highest result was FairFuzz displaying only a 12% improvement in comparison to AFL.

The authors claim that this work differs from previously available parallelized fuzzing systems by sharing metrics in real-time [94, p. 3]. Additionally, the installation instructions provided with source code are insufficient to produce a proper installation of EnFuzz. Trail of Bits has incorporated the ensemble fuzzing architecture (Figure 18), used by EnFuzz, into their fuzzing framework DeepState [98], [99].

5.6 Triage

5.6.1 Halfempty

Reference Link	https://github.com/googleprojectzero/halfempty
Target Type	Fuzzing Corpus
Host Operating System	Linux; macOS
Target Operating System	Linux; macOS
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	September 2018
License Type	Apache License v2.0
Maintenance	Maintained by Google Project Zero

Overview

Halfempty [100], is a test case minimization tool, developed by Google Project Zero, which is based on the delta debugging technique [101]. The goal is to minimize the size of an input file to facilitate analysis, thereby making it easier to identify the area of the input critical to triggering a bug. By properly tailoring the execution script, any new outcomes discovered during the minimization process can also be captured.

Fuzzing only produces inputs which cause crashes. The next step is understanding why, and what part of the input caused a crash. Halfempty uses delta debugging, a technique originally introduced by Andreas Zeller at Saarland University in 2000 [102] and later used in AFL's test case minimizer, afl-tmin [103]. Halfempty improves delta debugging and parallelizes the process, which vastly improves performance.

Design and Implementation

Halfempty requires a shell script that returns zero when given an input with the desired behavior (e.g., crashing the target application) and returns one otherwise. Halfempty then repeatedly runs this shell script to determine if a test case contains the desired behavior.

Halfempty uses “pessimistic speculative execution” to quickly identify a reduced input for test case minimization or delta debugging. In this case, pessimistic speculative execution means proactively testing branches under the assumption that the branches currently under test do not represent the smallest valid input. If these assumptions are correct, then the process will be substantially sped up. While this approach takes much less real-world time, it can result in a much larger amount of CPU time.



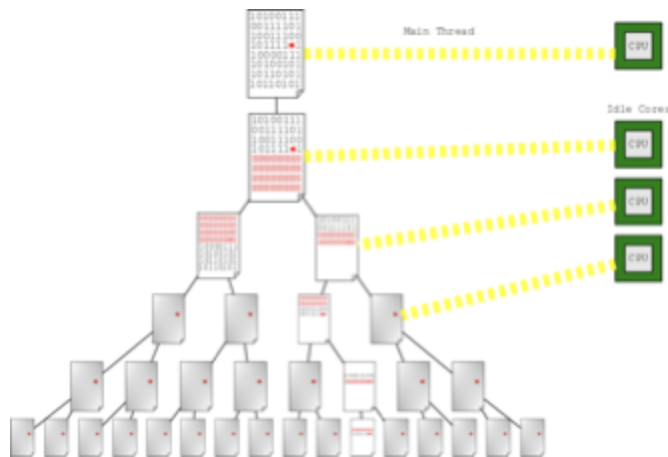


Figure 20: A binary tree of all possible outcomes, along with a plan for idle cores to speculatively work ahead of the main thread [100, p. "README.md"]

Use Cases and Limitations

Performing root cause analysis of a crash from a fuzzed input can sometimes be challenging due to the size of the mutated input file. Halfempty addresses this problem by removing bytes and overwriting mutated bytes with a specified byte, typically a null or whitespace. The process will run until the smallest and simplest file is identified that produces the same crash. This file can then be used in the root cause analysis.

Halfempty does however have limitations:

- Halfempty reduces file size by removing bytes and overwriting bytes with a specified byte. It does not have support for recalculating gatekeeper values such as length or checksum fields. However, a user may account for this limitation when developing the test shell script.
- Halfempty does not have built-in support for parallelism across a network. Again, a user may account for this limitation by developing a test shell script which enables inputs to be tested remotely.
- Halfempty only allows overwriting with a single specified byte, so it may encounter challenges on files with a mix of binary and text formats.
- There is a chance that Halfempty might be slower on single CPU systems, but most modern processors have multiple cores.

5.7 Trends

Speed

Until recently, the primary focus of fuzzer research has been on increasing coverage or in identifying fruitful code paths to target. However, new fuzzers have emerged which focus primarily on speed. Brandon Falk, a fuzzing researcher, recently gave a talk at the REcon conference on a fuzzer he built which takes advantage of the vectorized instructions in the Xeon

Phi processor [104]. He calls this technique *vectorized emulation* and estimates that it can emulate “approx (sic) 40 billion to 120 billion instructions per second.” On OpenBSD’s DHCP client he was able to fuzz 5 million test cases at 54 billion instructions per second. However, no source code or binary has been released, which is why this fuzzer was not one of the tools discussed in this section. Additionally, there are significant configuration and hardware requirements, meaning that using this fuzzer would require certain resources and expertise.

The area of fuzzing most afflicted by a lack of speed has been grammar-based fuzzers. Many of these generators have historically been slow, in many cases causing them to be a less effective (or even infeasible) option. However, the F1 Fuzzer (and Falk’s FZero implementation in Rust [89]) which was discussed in this section, presents a significant speed up over current grammar-based options. Both this, and Falk’s vectorized emulation fuzzer, represent a growing focus on improving speed in fuzzers.

Improved Instrumentation

Frizzer is an excellent example of applying new instrumentation methods to fuzzing by combining Frida and the generation-based fuzzer Radamsa [81] to create a coverage guided black box fuzzer. Other new research, such as the RetroWrite tool discussed in 3.13.3, would allow black box binaries to be statically instrumented for AFL. Other binary rewriting tools, like Multiverse (discussed in the previous report) also have similar potential.

Tools like AFL are easy to use out of the box, but they lack target-specific customization potential. honggfuzz allows the user to instrument a program for fuzzing more easily. Good instrumentation and configuration are important for coverage-based fuzzers.

Symbolic Execution and Fuzzing

Combining fuzzing with symbolic execution has been a well-known technique for years [76]. It garnered increased attention during the DARPA Cyber Grand Challenge [77] because many teams implemented their own hybrid fuzzers as part of their cyber reasoning systems. However, these implementations have either not been released, or have not been maintained (e.g., Driller). A recent fuzzer, QSYM [58] (discussed in the previous report), is an effective implementation of this hybrid fuzzing technique. Hopefully more, and improved versions, of these will be seen in the future.

Additionally, techniques informed by symbolic execution but that avoid using it have made their way into recent fuzzing research. One such example is the fuzzer Eclipser [105] (discussed in the previous report), which approximates path constraints rather than fully solving for them (which is a costly part of symbolic execution).

Combining Fuzzers

Trail of Bits recently released a fuzzing tool, DeepState [106] (discussed in the previous report), that would allow the user to combine multiple fuzzers for use on a target. The benefit is that, while fuzzers may be comparable in general quality, they may explore different paths and find



different bugs. The tool EnFuzz (featured in this section), goes one step further and combines the coverage information for five fuzzers (AFL, libFuzzer, AFLfast, Fairfuzz, and Radamsa) to create a fuzzer optimized for seed selection and mutation. Going forward, one might expect more fuzzing techniques to rely on the combination of several fuzzers, rather than expecting one fuzzer to act as a panacea.

New Evaluation Criteria

Since the release of *Evaluating Fuzz Testing* [78], new fuzzing research has begun to treat the evaluation criteria put forth in this paper as a standard. Fuzzers included in this report, such as EnFuzz and Retrowrite, have used their recommendations to measure their performance. This paper reflects a growing discussion in the fuzzing community around the need to create better measures of fuzzing performance.

6 Exploitation

6.1 Technical Overview

Program bugs come in many forms, but ones which could potentially be exploited to generate unintended effects are considered vulnerabilities. NIST SP 800-30 defines a vulnerability as “a weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source [107].” In vulnerability research, once a vulnerability has been identified, the next step is to effectively leverage that vulnerability with an exploit (i.e., to cause unintended behavior in the target). This is often a more challenging task than finding the original vulnerability. Almost all the tools discussed in the first two sections of this report are useful for both vulnerability discovery and exploitation. For example, a researcher might use a debugger to determine exactly how many ‘A’ characters to pad their payload with so that their exploit is correctly situated. By contrast, this section focuses on tools and techniques explicitly intended for exploitation. One of the tools, shadow, is a debugger designed specifically for exploiting heap vulnerabilities in programs that use the jemalloc allocator. It is common for exploitation tools or techniques to be designed for a specific kind of target.

Mitigations

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

- **Stack Canaries:** Values placed on the stack to prevent accessing memory past the end of the buffer.



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

Other mitigations and security practices have also proved useful. These include privilege separation and privilege revocation, which concerns, respectively, when parts of a program needing higher privileges are cordoned off from other, lower privilege parts a program, and when a process gives up its higher privileges when it no longer needs them. Other exploit prevention practices include proper sanitization of data input into a program.

The following discusses types of exploit tools and techniques that are actively being researched and developed.

Shellcode

Related EotA Tools: SynesthesiaYS

Shellcode is a set of binary instructions that will perform some kind of exploitative task, such as spawning a shell. However, while the name is derived from that example, shellcode can be designed to generate any number of effects. Shellcode often must be constrained to a certain input format consistent with the target program, such as using only alphanumeric bytes. This section discusses SynesthesiaYS [108], which is a shellcode generation tool that conforms the shellcode to definable encoding restrictions.

Control Flow Attacks

Mitigations such as non-executeable stacks (DEP/NX) prevent an attacker from writing shellcode to the stack, so return-oriented programming (ROP) is often used to circumvent this. ROP is a technique in which existing code is reused for malicious purposes. In its most basic form, small snippets of instructions that end in a return instruction can be chained together by pushing their addresses onto the stack, and with the right chain of instructions, an attacker can gain control by spawning a shell or some other task. These snippets of instructions are known as ‘gadgets,’ and with the right gadgets, ROP-like attacks can also be performed with other



branching instructions. There has been significant research into static analysis techniques to identify them within program binaries and libraries. ROP can be thought of as a technique to build a weird machine [109] in which valid parts of the program are manipulated to produce unintended behavior.

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

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

- Blind ROP (BROP): BROP is a technique for attacking remote proprietary services for which the binary is not available [110].
- Sigreturn-oriented programming (SROP): SROP manipulates signal handling in Unix systems to construct a ROP-like weird machine [111].
- JIT-ROP: JIT-ROP is a technique to use ROP with a Just-In-Time (JIT) compiler for JavaScript in browsers. JIT-ROP makes the JIT produces gadgets that can then be used for ROP [112].
- Data-oriented programming (DOP): A technique which creates a ROP-like weird machine only with non-control data in memory. This avoids CFI mitigations [113].
- Block-oriented programming (BOP): This technique builds on DOP to automate it [114].
- Jump-oriented programming (JOP): Creates a ROP-like weird machine with jump instructions rather than return instructions [115].
- Counterfeit object-oriented programming (COOP): A technique developed to avoid CFI mitigations by chaining together existing C++ virtual functions [116].

Heap Exploitation

Related EotA Tools: SEIVE and SHRIKE

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

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



- **Heap Overflow:** Like a stack overflow, this is enabled by a vulnerability that allows an attacker to write past the end of the intended buffer on the heap, and overwrite other areas in a malicious manner.
- **Heap Spraying:** This is a technique in which a payload is ‘sprayed’ onto the heap to increase the likelihood that it is in the correct place for the exploit to work.
- **Double-free:** This is a vulnerability in which a heap object is freed twice. An attacker can then possibly control a pointer to that area of memory in a malicious way.

There are many other ways to attack the heap. These kinds of exploitation techniques are often more complicated than stack-based attacks, and can require knowledge of the internals of a specific allocator. The tool shadow (discussed in this section) is specifically intended for the allocator jemalloc.

Automatic Exploit Generation (AEG)

Related EotA Tools: SEIVE and SHRIKE, Revery, angr

AEG techniques are an area of research which has existed for a decade but gained notoriety in 2016 during the DARPA Cyber Grand Challenge (CGC) [77]. As the name suggest, AEG involves automatically writing an exploit for a vulnerability. Sometimes the AEG tool is given a known vulnerability, but an end-to-end AEG machine is designed to automatically discover the vulnerability and then exploit it. Often, this can be conceived as a two-step process: first discovering an input which triggers the vulnerability, then generating a ‘payload’ to exploit that vulnerability [117, p. 11]. Many of the tools and techniques discussed elsewhere in this report and its previous version are applicable to AEG, especially symbolic execution and fuzzing.

This section focuses on the second step, automatic exploitation after vulnerability discovery. AEG techniques have been applied to the stack, heap, kernel, etc. The tools SHRIKE and SEIVE are AEG tools for heap-based attacks. The previous version of this report discussed the tool FUZE [118], which was an AEG tool designed to exploit the Linux kernel.

6.2 Heap

6.2.1 SIEVE and SHRIKE

Reference Link	https://github.com/SeanHeelan/HeapLayout
Target Type	PHP Interpreter
Host/Target Operating System	Linux
Host/Target Architecture	x86 (32, 64)
Initial Release	October 2018
License Type	Open-Source
Maintenance	Not Actively Maintained



Overview

SHRIKE and SIEVE are the heap layout manipulation tools presented in *Automatic Heap Layout Manipulation for Exploitation* [119], which applies Automatic Exploit Generation (AEG) to heap layout manipulation. The paper presents two tools [120]. SIEVE is a framework for evaluating or testing heap layout manipulation (HLM) search algorithms. SHRIKE, is a HLM system which targets the PHP interpreter, by taking a partial exploit that requires a specific heap layout and automatically completes the exploit by generating the desired alignments. Heelan et al. claim their paper as “the first automatic approach” to heap layout manipulation.

Design and Implementation

The goal is to separate the specific problem of heap layout manipulation (HLM) from the larger process of AEG (for heap exploits). Rather than a multifaceted process of vulnerability discovery and subsequent exploit generation, HLM can be summarized as a much smaller and targeted subprocess that “primarily consist[s] of two activities: creating and filling holes in memory [119, p. 3].” Once a heap-based vulnerability has been identified, an exploit of that vulnerability often requires a specific heap layout. This research asks, given some constraints, is there a way to manipulate the heap to get the desired layout. This problem is described by the authors in [119, p. 4]:

“Given the API for a target program and a means by which to allocate a source and destination buffer, find a sequence of API calls that position the destination and source at a specific offset from each other.”

Their problem is constrained in several ways: they assume a deterministic allocator, a known starting state of the heap, and that there are no outside interactions with the heap after this starting state. While this limits the reach of their tools, they were able to parlay their research in SEIVE into a real-world exploit generation tool, SHRIKE.

SEIVE: SEIVE is described as a driver which is linked with a deterministic allocator that exposes malloc, free, calloc and realloc functions. It takes input in the form of a candidate file that lists an iterative series of allocations and deallocations. There are six possible directives that could be in this input file [119, p. 5]:

- a) <malloc size ID>
- b) <calloc nmeb size ID>
- c) <free ID>
- d) <realloc oldID size ID>
- e) <frst size>
- f) <snd size>

While the first four are straightforward calls, (e) and (g) denote the destination and source allocations. The SEIVE driver then runs with this input file, and returns the distance between the source and destination buffer.



In the algorithm in Figure 21, a series of random candidate input are constructed for a set source and destination buffer, and then the function ‘Execute’ is called to run the SEIVE driver on each candidate input. In this way, SEIVE is leveraged by a black-box search algorithm to find specific series of allocations and deallocations that result in the source and destination buffers being an exact offset (distance) from each other.

Algorithm 1 Find a solution that places two allocations in memory at a specified distance from each other. The integer g is the number of candidates to try, d the required distance, m the maximum candidate size and r the ratio of allocations to deallocations for each candidate.

```

1: function SEARCH( $g, d, m, r$ )
2:   for  $i \leftarrow 0, g - 1$  do
3:      $cand \leftarrow \text{ConstructCandidate}(m, r)$ 
4:      $dist \leftarrow \text{Execute}(cand)$ 
5:     if  $dist = d$  then
6:       return  $cand$ 
7:   return None

8: function CONSTRUCTCANDIDATE( $m, r$ )
9:    $cand \leftarrow \text{InitCandidate}(\text{GetStartingState}())$ 
10:   $len \leftarrow \text{Random}(1, m)$ 
11:   $fstIdx \leftarrow \text{Random}(0, len - 1)$ 
12:  for  $i \leftarrow 0, len - 1$  do
13:    if  $i = fstIdx$  then
14:       $\text{AppendFstSequence}(cand)$ 
15:    else if  $\text{Random}(1, 100) \leq r$  then
16:       $\text{AppendAllocSequence}(cand)$ 
17:    else
18:       $\text{AppendFreeSequence}(cand)$ 
19:   $\text{AppendSndSequence}(cand)$ 
20:  return  $cand$ 

```

Figure 21: Black-box search algorithm implemented on top of the SEIVE driver [119, p. 5]

Once this input file is found, the exploit writer now knows a series of allocations and deallocations that will result in a layout in which the vulnerability can be exploited.

SHRIKE: SHRIKE applies SEIVE to the PHP interpreter by constructing a search algorithm which “takes in a template, parses it and then constructs and executes PHP programs until a solution is found or the execution budget expires [119, p. 8].” A template, shown in Figure 22, is made up of valid PHP code, and directives to SHRIKE tagged with #X-SHRIKE.



```

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

```

Listing 4: Exploit template for CVE-2013-2110

Figure 22: Example exploit template [119, p. 5]

These tags direct SHRIKE to inject PHP code which records the address of an allocation or checks the distance between two previous allocations.

The architecture diagram from SHRIKE is shown in Figure 23:

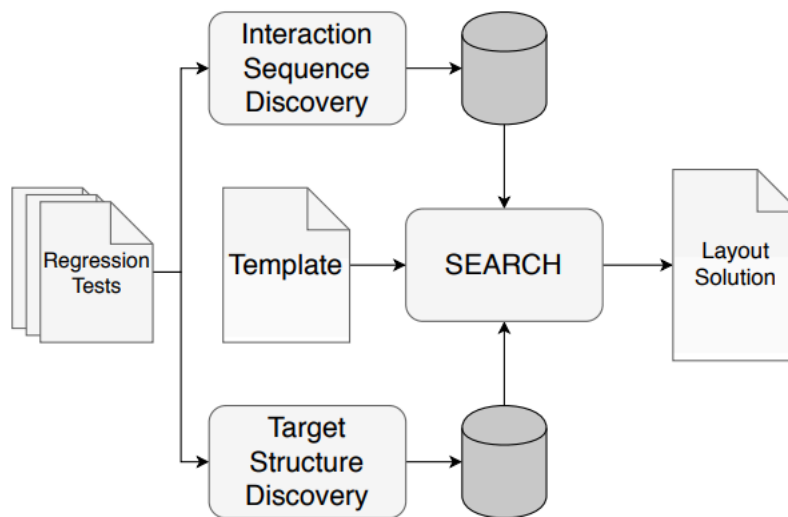


Figure 23: Example exploit template [119, p. 4]

Use Cases and Limitations

The obvious use case of SEIVE and SHRIKE is for heap exploit generation, specifically with regard to AEG. While automating control flow attacks has been researched, the heap has received less focus, in part because it is a more complicated target. This research also puts forth the idea of exploit templates, discussed further in Section 6.4, which could facilitate similar AEG research. This work is promising, but limited by the scope of these tools. By constraining these tools to a deterministic allocator, a known starting state of the heap, and that there are no outside interactions with the heap after this starting state, the real-world potential is limited. Additionally, as paper artifacts, these implementations are not mature, stable and maintained.



However, as it stands, this work does have real world attack potential. The authors show how SHRKE can be successfully leveraged against real CVEs in the PHP interpreter to hijack the control flow. The authors assert that,

“Our experiments with PHP indicate that automatic HLM can be performed effectively for real world programs...SHRIKE had a 70% success rate overall, and a 100% success rate in cases where there was no noise. SHRIKE demonstrates that it is possible to automate the process in an end-to-end manner, with automatic discovery of a mapping from the target program’s API to interaction sequences, discovery of interesting corruption targets, and search for the required layout. Furthermore, SHRIKE’s template based approach show that a system with these capabilities can be naturally integrated into the exploit development process. [119, p. 9]”

6.2.2 HeapHopper

Reference Link	https://github.com/angr/heaphopper
Target Type	Binary - Heap Implementations
Host Operating System	Linux
Target Operating System	Linux
Host Architecture	x86 (32, 64)
Target Architecture	x86 (32, 64)
Initial Release	2018
License Type	Open-Source
Maintenance	Maintained by angr

Overview

HeapHopper is a tool which uses symbolic execution to analyze the exploitability of heap implementations such as ptmalloc and dlmalloc. Described in the paper, *HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security* [121], HeapHopper models different combinations of heap vulnerabilities and benign heap interactions, and symbolically executes these models with a specific allocator to identify possible methods of exploitation. Once identified, these exploit primitives are converted into viable C source code implementations. HeapHopper was created by the developers of symbolic execution tool angr, and uses angr to allow HeapHopper to assess the security of an allocator. Several such allocators were analyzed: ptmalloc and standard dlmalloc had the worst security, however a security enhanced version of dlmalloc and musl’s allocator performed far better. HeapHopper can be useful for evaluating the efficacy of new security measures and identifying possible improvements, which was shown with a recent patch to ptmalloc. [121], [122]

Design and Implementation



HeapHopper was developed in response to a patch to ptmalloc which was designed to mitigate off-by-one NULL byte overwrites. The original motivations for this patch was an off-by-one NULL byte overwrite which was successfully leveraged into a full chunk overwrite. In designing this patch, the creator intended to prevent similar escalation from vulnerability to full exploit primitive. However, HeapHopper was able to identify a bypass to the patch, and use that data to construct a more effective mitigation. The authors describe HeapHopper as “the first approach to bring bounded model checking to the exploitability analysis of dynamic memory allocator implementations in the presence of memory corruption.”

HeapHopper targets specific heap implementations by taking as input an implementation as a shared object file. This is accompanied by a configuration file which prescribes the heap interactions to be tested. These two inputs are shown in Figure 24 which diagrams the workflow HeapHopper.

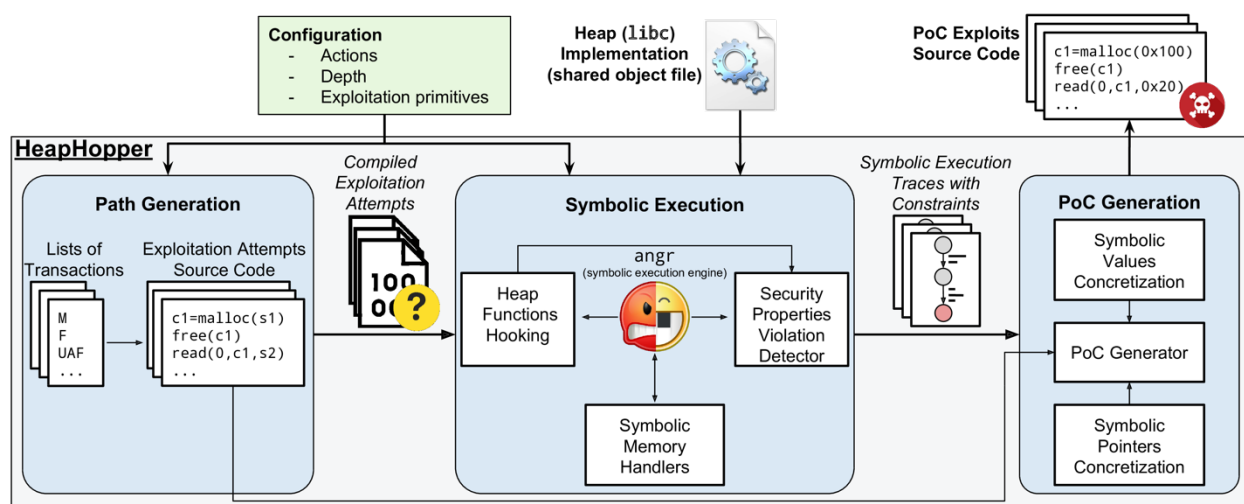


Figure 24: Overview of HeapHopper [121, p. 5]

The configuration file contains a list of allowed heap operations, termed *heap transactions*, a bound that sets maximum number of possible operations, and a list of exploit primitives that serve as the goals. Heap Hopper implements six possible heap transactions which the user can specify in the configuration file. These include not just benign heap operations, but vulnerable ones such as a transaction that causes an overflow. These transactions are [121, pp. 6-7]:

- **Malloc**
- **Free**
- **Overflow**
- **Use-After-Free**
- **Double-Free**
- **Fake-Free**

Each of these transactions is modeled with symbolic variables. Based on the transactions specified in the configuration file, HeapHopper combines these models into a sequence of

transactions, constructing almost all possible permutations (within some constraints and using the maximums also given in the configuration file). Each sequence corresponds with a potential path.

These sequences are then symbolically executed using the given allocator, and HeapHopper watches for security violations (i.e. exploit primitives) using instrumentation. Once an exploit primitive has been identified, HeapHopper then constructs viable C source code implementations of the path to achieve it, by concretizing the values identified through the symbolic execution. The possible security violations, or exploit primitives, that HeapHopper has implemented are [121, p. 8]:

- **Overlapping Allocation (OA)**
- **Non-Heap Allocation (NHA)**
- **Arbitrary Write (AW and AWC)**

Use Cases and Limitations

HeapHopper is intended to assess the security of heap implementations against common heap vulnerabilities. HeapHopper examined the allocators ptmalloc, dlmalloc and the musl implementation, but could theoretically be applied to more allocators, and potentially other operating systems. This tool has been shown to be a useful evaluation method for proposed patches to allocators, as such it can help determine whether a patch mitigates the intended vulnerability and what alternative or additional measures could be used. However, the exploits that HeapHopper produces are only intended to be proofs of concept. While the techniques used here could theoretically be used to identify an exploit for a given vulnerability, the constraints are so great that the practical viability is greatly diminished.

6.2.3 shadow

Reference Link	https://github.com/CENSUS/shadow
Target Type	Binary
Host/Target Operating System	Linux; Windows
Host/Target Architecture	x86 (32, 64); ARM (32, 64)
Initial Release	Version 2 Release: August 2017
License Type	Open-Source
Maintenance	Maintained by CENSUS Lab

Overview

shadow is a heap exploitation framework specifically for the jemalloc allocator, which is used most notably in Android and Firefox [123]. It is designed primarily for these two use cases, but could be generalized for generic jemalloc exploitation. It is designed to target the Windows



version of Firefox, and to do so it is tightly integrated with Windbg. It can also be used as a Python library. [124], [125]

Design and Implementation

shadow functions as a debugging tool, and operates on a target by tracking jemalloc calls and saving their metadata to a Python pickle file (enabling shadow to work faster rather than reparsing this state each time). Although it is designed to be agnostic of the target application, it is primarily designed for Android and Firefox on Windows; the latter application will be the focus in this report.

Windows Firefox with Windbg: The shadow plugin for Windbg comprises several features, which are accessed through commands. These include dumping info on various heap structures, parsing heap objects, and even modifying them. An abbreviated list of available commands and their descriptions, which is available to the user by calling `help`, is as follows:

Command	Description
<code>jechunks</code>	dump info on all available chunks
<code>jearenas</code>	dump info on jemalloc arenas
<code>jerun <address></code>	dump info on a single run
<code>jeruns</code>	dump info on jemalloc runs
<code>jebins</code>	dump info on jemalloc bins
<code>jebininfo</code>	dump info on bin sizes
<code>jeregions <size class></code>	dump all runs that host the regions of the given size class
<code>jechunks</code>	dump info on all available chunks
<code>jerearch <hex></code>	search the heap for the given hex dword
<code>jedump [path]</code>	store the heap snapshot to the current working directory or to the specified path
<code>jetcaches</code>	dump info on all teaches
<code>jeparse</code>	parse jemalloc structures from memory

Table 2: shadow commands [124]

Additionally, the following are Firefox specific commands:

Command	Description
<code>symbol <size></code>	display all Firefox symbols of the given size
<code>pa <address> [length]</code>	modify the ArrayObject's length (default new length 0x666)
<code>nursery</code>	display info on the SpiderMonkey GC nursery

Table 3: shadow commands for Firefox [124]

The first command listed in the previous table, **symbol**, is particularly effective for exploiting Use-After-Free (UAF) vulnerabilities, or positioning objects to overwrite or corrupt them.



Python Library: shadow can also be used as a Python library, which allows the user to access the functionality behind shadow's Windbg commands, in addition to other functionality.

Use Cases and Limitations

This debugger is mainly useful as a Firefox on Windows and Android heap exploitation tool, but its functionality can be generalized for other jemalloc uses. Its use case is like that of a standard debugger, but specialized for a jemalloc heap.

The tool is limited in scope to jemalloc, which is not the most frequently used allocator. It also requires the use of a specific Windows debugging tool, again limiting the user.

6.3 Shellcode

6.3.1 SynesthesiaYS

Reference Link	https://github.com/RolfRolles/SynesthesiaYS
Target Type	x86 Shellcode (designed to be extensible to other machine codes)
Host/Target Operating System	N/A
Host/Target Architecture	x86 (32)
Initial Release	July 2017
License Type	GNU GPL v3
Maintenance	Maintained by Rolf Rolles; Last commit on 07/17/2017

Overview

SynesthesiaYS is a shellcode generation tool that performs automated shellcode generation and transformation to produce shellcode conforming to definable encoding restrictions [126]. The tool leverages the Yices SMT solver to generate the shellcode based on statements representing shellcode instructions, shellcode goals, and encoding restrictions. The goal of SynesthesiaYS is to have a general, user-friendly tool that can generate encoding-conformant code from unencoded shellcode, a functionality description, or a program, without requiring the user to be an expert in SMT solving. It currently supports 32-bit x86 shellcode, but has been designed to support other assembly languages. [126], [127], [128], [129]

Design and Implementation

In its current state, SynesthesiaYS is a set of Yices scripts that can be customized and extended to generate shellcode conforming to different encodings. Lines of Yices script represent basic x86 assembly instructions and shellcode constraints including no NULL bytes, no '%' character, all ASCII bytes, and all uppercase characters.



SynesthesiaYS is meant to address issues with existing encoders that generated encoded shellcode with a decoder stub. The first issue is that these shellcode generators can greatly expand the shellcode size, breaking exploitation size restrictions. Second, existing encoders must be manually updated to work with new encoding restrictions. Third, intrusion detection system (IDS) signatures can detect common patterns used by decoders.

The author states that SynesthesiaYS has theoretical properties to address the issues with existing encoders. The tool will be fully automated, not requiring any manual analysis, flexible to support arbitrary encoding restrictions or processors, exhaustive, optimal, and having the ability to produce self-modifying code without using common patterns that an IDS could catch.

Use Cases and Limitations

“At present, the current implementation falls short of the goal of being an actual compiler: the process is not fully (or even largely) automated [128].” The developer, Rolf Rolles, also mentions in his slideshow that “Synesthesia is still a research idea with practical limitations [127].” Rolles stated in 2017 that he has a working version, still in early development, that is a stand-alone compiler using a custom general assembly language named SIMPLE that can be converted to x86 assembly. Unfortunately, he has not released an update since that statement was made over two years ago.

The output of the Yices program run on the SynesthesiaYS scripts must be manually interpreted, requiring some knowledge of the use of SMT solvers. The shellcode results are currently printed in binary format which would have to be converted to hexadecimal strings or another format more useful for exploitation scripts. For example, one run produced the following shellcode result:

```
# yices --mode=ef synesthesia-1-mod-longestsolution.js
sat
(= bytecode00 0b11100000)
(= bytecode01 0b00000001)
(= bytecode02 0b00000000)
(= bytecode03 0b00000000)
(= bytecode04 0b00000000)
(= bytecode05 0b11111000)
(= bytecode06 0b00000000)
(= bytecode07 0b00000000)
(= bytecode08 0b00000000)
...
```

Figure 25: Output of Yices program run on SynesthesiaYS script.

Creation of Yices scripts describing constraints can be done by modifying existing sample scripts provided in the GIT repository, but they still require an understanding of the Yices constraint syntax. The user must have knowledge of the SMT-LIB language to develop programs describing shellcode encoding limitations, the operating environment, and the shellcode goals. While the basics of the language might not be difficult to grasp, it should be noted that the language specification is not a quick read at 103 pages long.



The example files have repetitive lines that would be tedious to generate by hand. One example repeats “(define bytecode00::Byte)” and “(define shellcode00::Byte)” 128 times, “(define realsbyte00::Byte 0xA0)”, “(ite (= idx 0x00) realsbyte00)”, and “(ite (= idx 0x00) (bv-extract 7 0 sc))” 64 times, and “(ite (= x 0x00) bytecode00)” 256 times. While these lines would be simple to generate with short lines from a scripting language such as Python, Perl, or even Bash, a higher-level abstraction would be very helpful to improve readability and maintainability.

In its current state, SynesthesiaYS is a great research project and an initial step towards a more powerful shellcode generation and transformation tool. Improvements in the tool could provide much faster shellcode generation and the generation of shellcode that blends in better with normal target traffic, making it more difficult for defenders to detect.

6.4 Trends

Increasingly Sophisticated and Diverse AEG

AEG, although a topic of research for over a decade, has become increasingly sophisticated, in part driven by the DARPA Cyber Grand Challenge (CGC) in 2016. This is in part due to improvements in other vulnerability discovery and program analysis techniques like fuzzing and symbolic execution, but also due to the growing diversity of AEG techniques. AEG has recently been applied to heap exploitation and kernel exploitation, among other areas. This allows AEG tools to exploit a larger number of vulnerabilities that it identifies.

Exploit Templates

The tools SHRIKE and SEIVE, discussed in this section, are based on the paper *Automatic Heap Layout Manipulation for Exploitation* [120] by Heelan et al., which explores the idea of exploit templates. In a subsequent blog post [130], Heelan predicts that the “future of automatic exploit generation is going to look more like template-based approaches than end-to-end solutions”. Heelan defines an exploit template as follows:

“An exploit template is a simply a (sic) partially completed exploit where the incomplete parts are to be filled in by some sort of automated reasoning engine [130].”

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

“In ‘normal’ software development it is accepted that things like long build cycles are bad, while REPLs [Read-eval-print loop] are generally good. The reason is that the latter supports a tight loop of forming a hypothesis, testing it, refining and repeating, while the former breaks this process. Exploit writing has a similar hypothesis refinement loop and any technology that can make this loop tighter will make the process more efficient [130].”

If Heelan’s prediction is correct, moving forward one can expect to see more template-based AEG approaches, rather than full vulnerability discovery to exploit generation solutions.



7 Conclusion

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



8 References

- [1] S. Dinesh, N. Burow, D. Xu and M. Payer, "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization," in *IEEE Symposium on Security and Privacy*, San Francisco, CA, 2020.
- [2] Hex-Rays, "Hex-Rays," 30 09 2019. [Online]. Available: Hex-Rays.com. [Accessed 21 1 2020].
- [3] Vector 35, "Binary Ninja," 2019. [Online]. Available: binary.ninja. [Accessed 21 1 2020].
- [4] Free Software Foundation, Inc., "GNU Binutils," 12 10 2019. [Online]. Available: <https://www.gnu.org/software/binutils/>. [Accessed 21 1 2020].
- [5] N. A. Quynh, "Capstone Engine," [Online]. Available: <http://www.capstone-engine.org/>. [Accessed 21 1 2020].
- [6] Seclab at University of California, Santa Barbara and SEFCOM at Arizona State University, "angr," [Online]. Available: <http://angr.io/>. [Accessed 21 1 2020].
- [7] Qiling Framework, "Qiling Advanced Binary Emulation framework," [Online]. Available: <https://github.com/qilingframework/qiling>. [Accessed 21 01 2020].
- [8] O. A. V. Ravnås, "Frida," [Online]. Available: frida.re. [Accessed 21 01 2020].
- [9] CEA IT Security, "Miasm," [Online]. Available: miasm.re. [Accessed 21 1 2020].
- [10] National Security Agency, "Ghidra," [Online]. Available: ghidra-sre.org. [Accessed 21 1 2020].
- [11] S. Wang, P. Wang and D. Wu, "Reassembleable Disassembly," in *USENIX Security Symposium*, Washington, D.C, 2015.
- [12] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel and G. Vigna, "Ramblr: Making Reassembly Great Again," in *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2017.
- [13] A. Flores-Montoya and E. Schulte, "Datalog Disassembly," *eprint arXiv:1906.03969*, 2019.
- [14] Quarkslab, "LIEF," [Online]. Available: <https://lief.quarkslab.com/>. [Accessed 21 1 2020].
- [15] LLVM Foundation, "The LLVM Compiler Infrastructure," [Online]. Available: <https://llvm.org/>. [Accessed 21 01 2020].
- [16] LLVM Foundation, "Clang: a C language family frontend for LLVM," [Online]. Available: <https://clang.llvm.org/>. [Accessed 21 1 2020].
- [17] L. Foundation, "LLVM Language Reference Manual," [Online]. Available: <https://llvm.org/docs/LangRef.html>. [Accessed 21 1 2020].
- [18] LLVM Foundation, "'CLANG' CFE INTERNALS MANUAL," [Online]. Available: <https://clang.llvm.org/docs/InternalsManual.html>. [Accessed 21 1 2020].
- [19] National Security Agency, "Ghidra Software Reverse Engineering Framework," [Online]. Available: <https://github.com/NationalSecurityAgency/ghidra>. [Accessed 21 1 2020].



- [20] A. Bulazel, "WORKING WITH GHIDRA'S P-CODE TO IDENTIFY VULNERABLE FUNCTION CALLS," Riverloop Security, 11 05 2019. [Online]. Available: <https://www.riverloopsecurity.com/blog/2019/05/pcode/>. [Accessed 21 1 2020].
- [21] "P. LaFosse and J. Weins, "Modern Binary Analysis with ILs," presented at the BlueHat Seattle 2019, BlueHat Seattle 2019, 25-Oct-2019."
- [22] R. Rolles, "Hex-Rays Microcode API vs. Obfuscating Compiler," 19 09 2018. [Online]. Available: <https://www.hexblog.com/?p=1248>. [Accessed 21 1 2020].
- [23] Vector 35, "Binary Ninja Intermediate Language Series, Part 1: Low Level IL," [Online]. Available: <https://docs.binary.ninja/dev/bnil-llil.html>. [Accessed 21 1 2020].
- [24] "CMU Cylab, "Carnegie Mellon University Binary Analysis Platform (CMU BAP)." [Online]. Available: <https://github.com/BinaryAnalysisPlatform/bap>," [Online]. [Accessed 21 1 2020].
- [25] "CMU Cylab, "A formal specification for BIL: BIL Instruction Language," 02-Oct-2015. [Online]. Available: <https://github.com/BinaryAnalysisPlatform/bil/releases/download/v0.1/bil.pdf>," [Online]. [Accessed 10 1 2020].
- [26] Fabrice Desclaux, "Miasm : Framework de reverse engineering," *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC) 2012*, p. 25, 2012..
- [27] CEA IT Security, "Miasm." [Online]. Available: <https://github.com/cea-sec/miasm>. [Accessed: 10-Jan-2020]..
- [28] S. Groß, "FuzzIL: Coverage Guided Fuzzing for JavaScript Engines," Karlsruhe Institute of Technology (KIT), 2018..
- [29] Y. Shoshitaishvili et al., "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157, doi: 10.1109/SP.2016.17..
- [30] Google Project Zero, "WinAFL Dynamorio Instrumentation mode," 21-Jun-2019. [Online]. Available: <https://github.com/googleprojectzero/winafl>. [Accessed: 21-Jun-2019]..
- [31] GrammaTech, "GTIRB." [Online]. Available: <https://github.com/GrammaTech/gtirb/blob/master/README.md>. [Accessed: 10-Jan-2020]..
- [32] C. Cifuentes, "Reverse Compilation Techniques," *Queensland University of Technology*, 1994..
- [33] I. Guilfanov, "Keynote: The story of IDA Pro," presented at CODE BLUE 2014, 2014..
- [34] L. H. Newman, "The NSA Makes Ghidra, a Powerful Cybersecurity Tool, Open Source," *Wired*, 05-2019. [Online]. Available: <https://www.wired.com/story/nsa-ghidra-open-source-tool/>. [Accessed 2019 01 21]..
- [35] Synopsys, "COVERITY SCAN STATIC ANALYSIS," [Online]. Available: <https://scan.coverity.com/>. [Accessed 21 1 2020].
- [36] GrammaTech, "CodeSonar," [Online]. Available: <https://www.grammatech.com/products/codesonar>. [Accessed 21 1 2020].



- [37] Semmle, "Semmle - Code Analysis Platform for Securing Software," [Online]. Available:
] <https://semml.com/>. [Accessed 21 1 2020].
- [38] Gotovchits, Ivan, "[ANN] BAP 2.0 Release," *OCaml*, Nov-2019. [Online]. Available:
] <https://discuss.ocaml.org/t/ann-bap-2-0-release/4719..>
- [39] CEA IT Security, "Release v0.1.0," *Miasm*, 20-Dec-2018. [Online]. Available:
] https://miasm.re/blog/2018/12/20/release_v0_1_0.html. [Accessed: 10-Jan-2020]..
- [40] F. Desclaux and C. Mougey, "Miasm2 Reverse Engineering Framework," presented at the
] *BlackHat USA 2018*, 08-Aug-2018..
- [41] RPISEC, "Modern Binary Exploitation - Lab1C," 22 6 2015. [Online]. Available:
] <https://github.com/RPISEC/MBE/blob/master/src/lab01/lab1C>. [Accessed 21 1 2020].
- [42] CEA IT Security, "Miasm IR getting higher," *Miasm*, 16-Jan-2019. [Online]. Available:
] https://miasm.re/blog/2019/01/16/miasm_ir_getting_higher.html#more. [Accessed: 10-Jan-2020]..
- [43] CEA IT Security, "Playing with Dynamic symbolic execution," *Miasm*, 05-Oct-2017.
] [Online]. Available:
https://miasm.re/blog/2017/10/05/playing_with_dynamic_symbolic_execution.html.
[Accessed: 10-Jan-2020]..
- [44] C. Cylab, "Binary Analysis Platform," [Online]. Available:
] <http://binaryanalysisplatform.github.io/bap/api/odoc/index.html>. [Accessed 21 01 2020].
- [45] wtbw, "BAP: The Next-Generation Binary Analysis Platform (by David Brumley;
] successor to VINE)," Reddit, 09 08 2009. [Online]. Available:
https://www.reddit.com/r/ReverseEngineering/comments/98zlc/bap_the_nextgeneration_binary_analysis_platform/. [Accessed 21 01 2020].
- [46] GrammaTech, "GrammaTech Releasing Binary Analysis and Rewriting Interface into Open
] Source," Gramma Tech News, 04 2019. [Online]. Available:
<https://news.grammatech.com/grammatech-releasing-binary-analysis-and-rewriting-interface-into-open-source>. [Accessed 21 01 2020].
- [47] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite," Github, 2019. [Online].
] Available: <https://github.com/HexHive/retrowrite>. [Accessed: 08-Jan-2020].
- [48] O. Sandoval, "drgn," [Online]. Available: <https://github.com/osandov/drgn>. [Accessed 21 1
] 2020].
- [49] Quarkslab, "Triton - A DBA Framework," [Online]. Available: triton.quarkslab.com.
] [Accessed 21 1 2020].
- [50] D. Liew, C. Cadar, A. F. Donaldson and J. R. Stinnett, "Just fuzz it: solving floating-point
] constraints using coverage-guided fuzzing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, Tallinn, Estonia, 2019.
- [51] Free Software Foundation, Inc., "GDB: The GNU Project Debugger," [Online]. Available:
] <https://www.gnu.org/software/gdb/>. [Accessed 21 1 2020].
- [52] Intel, "Pin - A Dynamic Binary Instrumentation Tool," Intel , [Online]. Available:
] <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>.
[Accessed 21 1 2020].



- [53] D. Bruening, "DynamoRIO," [Online]. Available: <https://www.dynamorio.org/>. [Accessed 21 1 2020].
- [54] Google Project Zero, "WinAFL," 21-Jun-2019. [Online]. Available: <https://github.com/googleprojectzero/winafl>. [Accessed: 21-Jun-2019].
- [55] T. V. Developers, "Valgrind," [Online]. Available: <https://valgrind.org/>. [Accessed 21 1 2020].
- [56] V. Sharma, M. Whalen, S. McCamant and V. Willem, "Veritesting Challenges in Symbolic Execution of Java," *ACM SIGSOFT Software Engineering Notes*, vol. 42, pp. 1-5, 2018.
- [57] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.
- [58] sslab-gatech, QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. [gts3.org \(SSLab@Gatech\)](https://github.com/SSLab@Gatech/qsym), 2019. .
- [59] H. Xu, Z. Zhao, Y. Zhou and M. R. Lyu, "On Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs," *On Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs*, December 2017.
- [60] M. Brain, F. Schanda and Y. Sun, "Building Better Bit-Blasting for Floating-Point Problems," in *TACAS 2019: Tools and Algorithms for the Construction and Analysis of Systems*, 2019.
- [61] O. Sandoval, "drgn," [Online]. Available: <https://drgn.readthedocs.io/en/latest/>. [Accessed 21 01 2020].
- [62] F. S. Foundation, "3.10 Options for Debugging Your Program," [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html>. [Accessed 22 1 2020].
- [63] K. J. Lua and A. Q. Nguyen, "Qiling," [Online]. Available: <https://www.qiling.io>. [Accessed 21 1 2020].
- [64] F. Soudel and J. Salwan, "Triton: Concolic Execution Framework," presented at the Symposium on Information and Communications Security (SSTIC), Jun-2015.
- [65] J. Salwan, "Triton under the hood," Quarkslab, [Online]. Available: <https://blog.quarkslab.com/triton-under-the-hood.html>. [Accessed 21 1 2020].
- [66] J. Salwan, "What kind of semantics information Triton can provide?," Quarkslab, [Online]. Available: <https://triton.quarkslab.com/blog/What-kind-of-semantics-information-Triton-can-provide/>. [Accessed 21 1 2020].
- [67] J. Salwan, Github, [Online]. Available: <https://github.com/JonathanSalwan/Triton>. [Accessed 21 1 2020].
- [68] Z3Prover, "Z3," Github, [Online]. Available: <https://github.com/Z3Prover/z3>. [Accessed 21 1 2020].
- [69] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "JFS," Multicore Group, Imperial College London. [Online]. Available: <https://github.com/mc-imperial/jfs>.
- [70] LLVM Foundation, "libFuzzer – a library for coverage-guided fuzz testing.," [Online]. Available: <https://llvm.org/docs/LibFuzzer.html#libfuzzer-a-library-for-coverage-guided-fuzz-testing>. [Accessed 22 1 2020].
- [71] Fondazione Bruno Kessler and DISI-University of Trento., "MathSAT 5," [Online]. Available: <http://mathsat.fbk.eu/>. [Accessed 22 1 2020].



- [72 M. Zalewski, "afl," [Online]. Available: <http://lcamtuf.coredump.cx/afl/>.
]
- [73 Google, "honggfuzz," [Online]. Available: <https://google.github.io/honggfuzz/>. [Accessed
] 21 1 2020].
- [74 D. Mantz and B. Kauer, "Frizzer," Github, [Online]. Available:
] <https://github.com/demantz/frizzer/>. [Accessed 21 2 2020].
- [75 H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-Aware Code Generation to
] Find Vulnerabilities in JavaScript Engines," in Proceedings 2019 Network and Distributed
System Security Symposium, San Diego, CA, 2019, doi: 10.14722/ndss.2019.23263.
- [76 P. Godefroid, M. Y. Levin, D. Molnar and Microsoft, "SAGE: Whitebox Fuzzing for
] Security Testing," *acmqueue*, vol. 10, no. 1, 11 January 2012.
- [77 Defense Advanced Research Project Agency (DARPA), "Cyber Grand Challenge (CGC)
] (Archived)," [Online]. Available: <https://www.darpa.mil/program/cyber-grand-challenge>.
[Accessed 22 1 2020].
- [78 G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in
] Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications
Security - CCS '18, Toronto, Canada, 2018, pp. 2123–2138, doi:
10.1145/3243734.3243804.
- [79 Google, "PersistentFuzzing," Google, [Online]. Available:
] <https://github.com/google/honggfuzz/blob/master/docs/PersistentFuzzing.md>. [Accessed 21
1 2020].
- [80 Google, "honggfuzz," Github, [Online]. Available: <https://github.com/google/honggfuzz>.
] [Accessed 21 01 2020].
- [81 A. Helin, "Radamsa," PROTOS Protocol Genome Project, [Online]. Available:
] <https://gitlab.com/akihe/radamsa>. [Accessed 21 2 2020].
- [82 bananamafia, "In-Process Fuzzing With Frida," [Online]. Available:
] <https://bananamafia.dev/post/frida-fuzz/>. [Accessed 22 1 2020].
- [83 C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing
] with input-to-state correspondence," in Proceedings of the Network and Distributed System
Security Symposium, 2019.
- [84 S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-
] assisted feedback fuzzing for OS kernels," in 26th USENIX Security Symposium
(USENIX Security 17). Vancouver, BC: USENIX Association, 2017, pp. 167–182. .
- [85 C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "NDSS 2019
] REDQUEEN: Fuzzing with Input-to-State Correspondence," presented at the Network and
Distributed System Security Symposium (NDSS), 2019.
- [86 RUB-SysSec, "https://github.com/RUB-SysSec/redqueen," Github, [Online]. Available:
] REDQUEEN. [Accessed 1 21 2020].
- [87 R. Gopinath and A. Zeller, "Building Fast Fuzzers," arXiv:1911.07707 [cs], Nov. 2019.
]
- [88 [1]R. Gopinath, "F1 Fuzzer." [Online]. Available: <https://pypi.org/project/F1-Fuzzer/>.
] [Accessed: 10-Jan-2020]. .



- [89] B. Falk, "fzero fuzzer," Github. [Online]. Available:
] https://github.com/gamozolabs/fzero_fuzzer. [Accessed: 10-Jan-2020].
- [90] J. Hertz and T. Newsham, "Project Triforce," NCC Group, [Online]. Available:
] <https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/>. [Accessed 13 12 2019].
- [91] Google, "Syzkaller - kernel fuzzer," Github, 2015. [Online]. Available:
] <https://github.com/google/syzkaller>. [Accessed 13 12 2019].
- [92] M. Tingu, "kAFL," 2019. [Online]. Available: <https://github.com/MatthewTingu/kAFL>.
] [Accessed 13 12 2019].
- [93] T. Newsham, "kAFL," 2017. [Online]. Available: <https://github.com/timnewsham/kAFL>.
] [Accessed 13 12 2019].
- [94] Y. Chen et al., "EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse
] Fuzzers," in Proceedings of the 28th USENIX Security Symposium., Santa Clara, California, 2019, p. 18.
- [95] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as
] Markov Chain," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16, Vienna, Austria, 2016, pp. 1032–1043, doi: 10.1145/2976749.2.
- [96] C. Lemieux and K. Sen, "FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox
] Fuzz Testing Coverage," Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018, pp. 475–485, 2018, doi: 10.1145/323814.
- [97] enfuzz, "enfuzz," Github, [Online]. Available: <https://github.com/enfuzz/enfuzz>. [Accessed
] 22 1 2020].
- [98] A. Cao, "DeepState Now Supports Ensemble Fuzzing," Trail of Bits, 3 09 2019. [Online].
] Available: <https://blog.trailofbits.com/2019/09/03/deepstate-now-supports-ensemble-fuzzing/>. [Accessed 22 1 2020].
- [99] Trail of Bits, "DeepState," Github, [Online]. Available:
] <https://github.com/trailofbits/deepstate>. [Accessed 22 1 2020].
- [10] G. P. Zero, "halfempty," Github, [Online]. Available:
0] <https://github.com/googleprojectzero/halfempty>. [Accessed 22 01 2020].
- [10] A. Zeller. [Online]. Available: <https://www.st.cs.uni-saarland.de/dd/>. [Accessed 21 1 2020].
1]
- [10] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?," *Lecture Notes in*
2] *Computer Science (LNCS)*, vol. 1687, pp. 253-267, September 1999.
- [10] GRIMM, "Delta Debugging," GRIMM, 20 6 2018. [Online]. Available: [https://blog.grimm-](https://blog.grimm-3)
3] [co.com/post/delta-debugging/](https://blog.grimm-co.com/post/delta-debugging/). [Accessed 22 1 2020].
- [10] B. Falk, "Vectorized Emulation: Hardware accelerated taint tracking at 2 trillion
4] instructions per second," 14 19 2018. [Online]. Available:
https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html. [Accessed 7 1
2020].
- [10] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box Concolic Testing on Binary Code," in
5] Proceedings of the 41st International Conference on Software Engineering, Piscataway, NJ, USA, 2019, pp. 736–747, doi: 10.1109/ICSE.2019.00082.



- [10] [1] Trail of Bits, "Deep State," Github, 21-Jun-2019. [Online]. Available:
6] <https://github.com/trailofbits/deepstate>. [Accessed: 21-Jun-2019]. .
- [10] Joint Task Force Transformation Initiative. September 2012. Guide for Conducting Risk
7] Assessments. In NIST Special Publication 800-30 Revision 1. U.S. Department of
Commerce, National Institute of Standards and Technology, Gaithersburg, MD. .
- [10] R. Rolles, "SynesthesiaYS," 17 07 2017. [Online]. Available:
8] <https://github.com/RolfRolles/SynesthesiaYS>. [Accessed 21 1 2020].
- [10] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman and A. Shubina, "Exploit
9] Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation,"
;login.;, December 2011 .
- [11] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in 2014
0] IEEE Symposium on Security and Privacy, 2014, pp. 227–242, doi: 10.1109/SP.2014.22.
- [11] E. Bosman and H. Bos, "Framing Signals - A Return to Portable Shellcode," in 2014 IEEE
1] Symposium on Security and Privacy, 2014, pp. 243–258, doi: 10.1109/SP.2014.23.
- [11] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "Just-In-
2] Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout
Randomization," in 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, 2013,
pp. 574.
- [11] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented
3] Programming: On the Expressiveness of Non-control Data Attacks," in 2016 IEEE
Symposium on Security and Privacy (SP), San Jose, CA, 2016, pp. 969–986, doi:
10.1109/SP.2016.
- [11] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block Oriented Programming:
4] Automating Data-Only Attacks," in Proceedings of the 2018 ACM SIGSAC Conference on
Computer and Communications Security, Toronto Canada, 2018, pp. 1868–1882, doi:
10.114.
- [11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class
5] of code-reuse attack," in Proceedings of the 6th ACM Symposium on Information,
Computer and Communications Security - ASIACCS '11, Hong Kong, China, 2011, p. 30.
- [11] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit
6] Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++
Applications," in 2015 IEEE Symposium on Security and Privacy, San Jose, CA.
- [11] S. D'Antoine, "Automatic Exploit Generation," presented at the CanSec West 2016, 2016. .
7]
- [11] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards Facilitating
8] Exploit Generation for Kernel Use-After-Free Vulnerabilities," in USENIX Security
Symposium, 2018. .
- [11] S. Heelan, T. Melham, and D. Kroening, "Automatic Heap Layout Manipulation for
9] Exploitation," in Proceedings of the 27th USENIX Conference on Security Symposium,
Berkeley, CA, USA, 2018, pp. 763–779.
- [12] S. Heelan, "HeapLayout," Github, [Online]. Available:
0] <https://github.com/SeanHeelan/HeapLayout>. [Accessed 21 1 2020].



- [12 M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Bringing
1] Bounded Model Checking to Heap Implementation Security," p. 19. .
- [12 angr, "HeapHopper," Github, [Online]. Available: <https://github.com/angr/heaphopper>.
2] [Accessed 30 1 2020].
- [12 CENSUS, "shadow," Github. [Online]. Available: <https://github.com/CENSUS/shadow>. .
3]
- [12 argp, "shadow :: De Mysteriis Dom jemalloc." [Online]. Available:
4] <https://github.com/CENSUS/shadow/commits/master/README.md>. .
- [12 argp, "OR'LYEH? The Shadow over Firefox." [Online]. Available:
5] <http://www.phrack.org/issues/69/14.html>. .
- [12 R. Rolles, "SynesthesiaYS," Github, [Online]. Available:
6] <https://github.com/RolfRolles/SynesthesiaYS>. [Accessed 22 1 2020].
- [12 R. Rolles, "Synesthesia: A Modern Approach to Shellcode Generation," Mobius Strip
7] Reverse Engineering, 2016. [Online]. Available:
<https://static1.squarespace.com/static/53a64cc2e4b0c63fc41a3320/t/582238ace6f2e15bc0f950c3/1478637750118/Synesthesia-Slides.pdf>. [Accessed 21 1 2020].
- [12 R. Rolles, "THE SYNESTHESIA SHELLCODE GENERATOR: CODE RELEASE AND
8] FUTURE DIRECTIONS," MÖBIUS STRIP REVERSE ENGINEERING, 17 07 2017.
[Online]. Available: <https://www.msreverseengineering.com/blog/2017/7/15/the-synesthesia-shellcode-generator-code-release-and-future-directions>. [Accessed 22 1 2020].
- [12 R. Rolles, "SYNESTHESIA: MODERN SHELLCODE SYNTHESIS (EKOPARTY 2016
9] TALK)," MÖBIUS STRIP REVERSE ENGINEERING, 8 11 2016. [Online]. Available:
<https://www.msreverseengineering.com/blog/2016/11/8/synesthesia-modern-shellcode-synthesis-ekoparty-2016-talk>. [Accessed 22 1 2020].
- [13 S. Heelan, "Automation in Exploit Generation with Exploit Templates," 5 3 2019. [Online].
0] Available: <https://sean.heelan.io/2019/03/05/automation-in-exploit-generation-with-exploit-templates/>. [Accessed 22 1 2020].
- [13 V. 35, "Stable Branch Changelog," [Online]. Available: <https://binary.ninja/changelog/>.
1] [Accessed 21 1 2020].
- [13 Avast, "RetDec," 2017. [Online]. Available: <https://retdec.com/>. [Accessed 21 01 2020].
2]
- [13 J. Weins, "State Of The Ninja: Episode 12," 02 01 2019. [Online]. Available:
3] <https://binary.ninja/2019/01/02/state-of-the-ninja-ep12.html>. [Accessed 21 01 2020].
- [13 Hex-Rays, "IDA: What's new in 7.3," Hex-Rays. [Online]. Available: <https://www.hex-rays.com/products/ida/7.3/index.shtml>. [Accessed: 21-Jun-2019].
4]
- [13 GrammaTech, "GTIRB." [Online]. Available: <https://github.com/GrammaTech/gtirb/>.
5] [Accessed: 10-Jan-2020]. .
- [13 D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "JFS," Multicore Group, Imperial
6] College London. [Online]. Available: <https://github.com/mc-imperial/jfs>. .
- [13 Microsoft, "Debugging Using WinDbg Preview," 12 1 2020. [Online]. Available:
7] <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-using-windbg-preview>. [Accessed 21 1 2020].



- [13] radareorg, "radare2," [Online]. Available: <https://github.com/radareorg/radare2>. [Accessed 8] 22 1 2020].
- [13] R.-S. /, "kAFL," 2017. [Online]. Available: <https://github.com/RUB-SysSec/kAFL>. 9] [Accessed 22 1 2020].
- [14] M. Zalewski, "releases," [Online]. Available: <http://lcamtuf.coredump.cx/afl/releases/>. 0] [Accessed 22 1 2020].
- [14] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," arXiv:1803.01307 1] [cs], Mar. 2018.
- [14] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox 2] Fuzzing," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17, Dallas, Texas, USA, 2017, pp. 2329–2344, doi: 10.1145/3133956.3134.
- [14] M. Zalewski, "Technical "whitepaper" for afl-fuzz," [Online]. Available: 3] http://lcamtuf.coredump.cx/afl/technical_details.txt. [Accessed 21 1 2020].
- [14] MITRE, "CWE-125: Out-of-bounds Read," [Online]. Available: 4] <https://cwe.mitre.org/data/definitions/125.html>. [Accessed 22 01 2020].
- [14] MITRE, "CWE-787: Out-of-bounds Write," [Online]. Available: 5] <https://cwe.mitre.org/data/definitions/787.html>. [Accessed 22 01 2020]..
- [14] MITRE, "CWE-476: NULL Pointer Dereference," [Online]. Available: 6] <https://cwe.mitre.org/data/definitions/476.html>. [Accessed 22 01 2020]..
- [14] MITRE, "CWE-401: Missing Release of Memory after Effective Lifetime," [Online]. 7] Available: <https://cwe.mitre.org/data/definitions/401.html>. [Accessed 22 01 2020]..
- [14] MITRE, "CWE-416: Use After Free," [Online]. Available: 8] <https://cwe.mitre.org/data/definitions/416.html>. [Accessed 22 01 2020]..
- [14] MITRE, "CWE-415: Double Free," [Online]. Available: 9] <https://cwe.mitre.org/data/definitions/415.html>. [Accessed 22 01 2020]..
- [15] MITRE, "CWE-697: Incorrect Comparison," [Online]. Available: 0] <https://cwe.mitre.org/data/definitions/697.html>. [Accessed 22 01 2020]..
- [15] MITRE, "CWE-768: Incorrect Short Circuit Evaluation," [Online]. Available: 1] <https://cwe.mitre.org/data/definitions/768.html>. [Accessed 22 01 2020]..
- [15] MITRE, "CWE-190: Integer Overflow or Wraparound" [Online]. Available: 2] <https://cwe.mitre.org/data/definitions/190.html>. [Accessed 22 01 2020]..
- [15] MITRE, "CWE-682: Incorrect Calculation" [Online]. Available: 3] <https://cwe.mitre.org/data/definitions/682.html>. [Accessed 22 01 2020]..
- [15] MITRE, "CWE-20: Improper Input Validation," [Online]. Available: 4] <https://cwe.mitre.org/data/definitions/20.html>. [Accessed 22 01 2020]..

