Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs

Jayakrishna Vadayath*, Moritz Eckert†, Kyle Zeng*, Nicolaas Weideman‡, Gokulkrishna Praveen Menon*, Yanick Fratantonio+, Davide Balzarotti†, Adam Doupé*, Tiffany Bao*, Ruoyu Wang*, Christophe Hauser‡, Yan Shoshitaishvili*

*Arizona State University, †EURECOM, ‡University of Southern California, +Cisco Systems Inc.

*{jvadayat,zengyhkyle,gpraveen,doupe,tbao,fishw,yans}@asu.edu †{moritz.eckert,davide.balzarotti}@eurecom.fr

+yanick.fratantonio.me ‡hauser@isi.edu ‡nweidema@usc.edu

Abstract

In spite of their effectiveness in the context of vulnerability discovery, current state-of-the-art binary program analysis approaches are limited by inherent trade-offs between accuracy and scalability. In this paper, we identify a set of vulnerability properties that can aid both static and dynamic vulnerability detection techniques, improving the precision of the former and the scalability of the latter. By carefully integrating static and dynamic techniques, we detect vulnerabilities that exhibit these properties in real-world programs at a large scale.

We implemented our technique, making several advancements in the analysis of binary code, and created a prototype called ARBITER. We demonstrate the effectiveness of our approach with a large-scale evaluation on four common vulnerability classes: CWE-131 (Incorrect Calculation of Buffer Size), CWE-252 (Unchecked Return Value), CWE-134 (Uncontrolled Format String), and CWE-337 (Predictable Seed in Pseudo-Random Number Generator). We evaluated our approach on more than 76,516 x86-64 binaries in the Ubuntu repositories and discovered new vulnerabilities, including a flaw inserted into programs during compilation.

1 Introduction

In spite of the constant evolution of security mechanisms and safeguards that were introduced in compilers, operating systems, and development environments, software vulnerabilities continue to be discovered. As a partial mitigation for this, the process of post-development analysis and testing has become standard practice in assessing the security of standard libraries, OS components, and embedded firmware alike.

In recent years, the state of the art in binary vulnerability discovery advanced with a panoply of new *dynamic* approaches, with a focus on fuzzing techniques. While this emphasis on fuzzing has led to dramatic improvements over existing techniques, and, therefore, represents a valuable path forward, it has come at a cost of investment in *static analysis*. This trade-off has downsides: "deep bugs", i.e., bugs "buried" deep within a program's possible execution paths or requiring intricate constraints tend to challenge dynamic techniques due to the dynamic coverage problem. Another dynamic approach, dynamic symbolic execution (DSE), is commonly used to exhaust execution paths within a small region of binary code and look for property violations. DSE's high-fidelity execution is both a bless and a curse: it also suffers from the dynamic coverage problem due to poor scalability on real-world binaries.

While revisiting static analysis would allow researchers to sidestep the dynamic coverage problem, current static analysis techniques on binary code lack the precision required to effectively identify vulnerabilities without overwhelming human analysts with false positives.

Vulnerability discovery techniques are most useful if they can be applied on a *wide* spectrum of programs in a fully *automated* manner without introducing overwhelming numbers of false positives. A *hybrid* approach—taking the best from both static and dynamic worlds—with high precision while maintaining high scalability, would be a powerful tool.

In this paper, we describe our effort to build such a tool, inspired by the evolution of fuzzing research. Because fuzzers must execute the target program, individual techniques are tailored to the analysis of individual program classes, such as kernel modules [20, 43] or language runtimes [19]. We realized that an analog to this concept in static analysis is the tailoring of static analyses to *specific vulnerability types*. By leveraging this insight, we identified a set of vulnerability properties that allow us to maintain, at the same time, a scalable (albeit imprecise) static detection and a precise (albeit less scalable) dynamic filtering of false positives. ARBITER is our hybrid analysis technique that can scalably analyze large amounts of binary code while maintaining high precision even in the case of complex vulnerabilities such as intricate occurrences of integer overflows or privilege escalation bugs. ARBITER is expandable and supports the specification of different vulnerability classes that exhibit the properties we identified, with new vulnerability class specifications requiring on the order of 75 lines of code. We present examples for four cases: CWE-131 (Incorrect Calculation of Buffer Size), CWE-252 (Unchecked Return Value), CWE-337 (Predictable Seed in PRNG), and CWE-134 (Uncontrolled Format String).

To achieve ARBITER's hybrid analysis, we introduce novel improvements to, and create novel combinations of, several binary-level analysis techniques, culminating in an adaptive false positive filtering step that uses static and dynamic techniques for a configurable trade-off between precision and performance. We also show how previous approaches—including those that target specific categories of bugs—are all affected

by limitations that hinder their precision and scalability. In fact, while one may think that detecting these vulnerabilities is trivial, we show that they are very challenging to identify in complex, real-world scenarios, and non-trivial even on "toy" code. For example, in an experiment on the synthetic Juliet dataset, ARBITER identified, and we manually confirmed, *190 vulnerabilities* in testcases that were erroneously considered safe by the ground truth and former analyses.

We evaluate ARBITER on 76,516 binary programs, which are collected from x86-64 Ubuntu 18.04 software repositories. We also demonstrate its precision by analyzing the 436 CWE-131 alerts that ARBITER raises in 366 programs, the 159 CWE-252 alerts across 126 programs, the 158 CWE-134 alerts across 119 programs, and the 377 CWE-337 alerts across 370 programs. These results demonstrate that ARBITER scales to real-world scenarios, and can detect bugs, including 0-day vulnerabilities, in real-world software. For example, we found and reported an exploitable vulnerability (CVE-2018-18311) in the Perl runtime and a heap error that affects *all 32-bit programs compiled by the OCaml compiler*.

Contributions. Our paper makes the following contributions:

- We identify a specific set of vulnerability properties that allow for the effective combination of static analysis and dynamic analysis, especially DSE, to achieve precision while maintaining scalability.
- We develop ARBITER, a framework that combines static analysis and DSE to identify bugs. ARBITER operates without any requirement for source code or build systems, and includes novel improvements to both static and dynamic techniques. Creating specifications of new vulnerability classes in ARBITER is inexpensive.
- We perform a large-scale evaluation of ARBITER, analyzing 76,516 binaries for four bug classes.

To support open research, ARBITER and data is available[1].

2 Related Work and Motivations

Table 1 lists six features that, from our view, are critical for real-world adoption of a vulnerability discovery technique. While much research focuses on vulnerability discovery, none of them provides an automated, scalable, and generic solution, which is the goal of this research. We first discuss research work that attempted (and failed) to achieve our research goal, then we present our observations and insights that led to the birth of ARBITER.

2.1 Vulnerability Discovery Techniques

We group existing vulnerability discovery techniques (summarized in Table 1) that are closely related to ARBITER along three main areas and discuss their advantages and limitations, which prevented them from being widely adopted.

[1] https://github.com/jkrshnmenon/arbiter

Tool	No Source	High Scalability	Low False Positive	No Harness	Symbolic Reasoning	Generic Vulns
Vanguard	✗	✓	✗	✓	✗	✗
Joern	✗	✓	✗	✓	✗	✓
CodeQL	✗	✓	✗	✓	✗	✓
Infer	✗	✓	✗	✓	✗	✓
AFL	✓	✗	✓	✗	✗	✓
Smartfuzz	✓	✗	✓	✗	✓	✗
DIODE	✓	✗	✓	✗	✓	✗
Statsym	✓	✗	✓	✗	✓	✓
IntScope	✓	✓	✓	✓	✓	✗
INDIO	✓	✓	✓	✓	✓	✗
ARBITER	✓	✓	✓	✓	✓	✓[2]

Table 1: Comparison between ARBITER and related tools in terms of their capabilities. We note whether the tool *requires source code*, has *high scalability* through the use of static techniques, has a *low false positive rate* through the use of dynamic techniques, *requires a harness* to execute the code under test, has *symbolic reasoning* capabilities, and supports *generic vulnerability types*.

White-box Static Vulnerability Analysis. Many techniques exist to find vulnerabilities in source code. However, there are fundamental challenges that differentiate static analysis from solutions that focus on binaries. Therefore, we only discuss approaches that are more closely related to ARBITER.

Graph-based vulnerability discovery approaches, such as Joern, Chucky, and CodeQL, rely on a number of carefully crafted queries that express patterns over a graph representation of the program source code [35, 44, 46]. Their main goal is to reduce the scope for a human analyst to only the potentially vulnerable parts of the code. As such, they do not try to be fully automated nor precise.

Some static tools are designed to find the same types of errors as ARBITER. For example, Vanguard [40] tries to identify missing security-sensitive checks by combining static code analysis and taint analysis. On top of requiring source code, Vanguard explicitly focuses on checks of operations directly affected by user input, and could not detect, among others, the failed dropped privilege vulnerability (shown in Section 3.2).

Dynamic Analysis on Binaries. There is a rich literature on the use of static analysis and DSE to drive the test case generation in a fuzzer [1, 2, 7, 27, 30, 37, 41, 49, 50].

SmartFuzz combines fuzzing and DSE to identify integer bugs in x86 Linux programs [25]. In addition to generating new test cases by using DSE, SmartFuzz uses a constraint solver to further identify assertion errors in integer operations, such as arithmetic overflows, non-value preserving width conversions, and dangerous signed/unsigned conversions. However, similar to other dynamic techniques, the scalability of SmartFuzz is severely limited by the symbolic-supported fuzzy exploration performed from the program entry point. Furthermore, SmartFuzz is limited to integer bugs, and it is unclear how much effort is required to support detection of more bug types.

[2] ARBITER can identify different types of vulnerabilities provided that they satisfy certain properties as described in Section 3.

Other solutions adopt taint analysis or DSE to uncover vulnerabilities in applications, e.g., Statsym [48], which uses statistics-guided DSE, and DIODE [39], which uses taint analysis to identify code locations that allocate memory and DSE to check if the integer argument of these allocation is vulnerable to overflow. While these solutions can find some vulnerabilities that ARBITER finds, both approaches require test-cases that exercise vulnerable points in target programs. In practice, this often requires an extensive fuzzing campaign.

Binary Static Analysis enhanced by DSE. Combining static analysis and DSE to verify results and reduce false positives is not novel. INDIO [51] is a pattern-matching solution that identifies and ranks code locations for potential vulnerabilities, then uses DSE with path pruning to validate the vulnerability. INDIO always starts from the program's entry point when performing DSE, which greatly limits its scalability.

IntScope [42] uses path-sensitive data-flow analysis to identify integer overflows, and taint analysis followed by DSE to verify that the overflow constraints are satisfied. The combination of static analysis and DSE and the ability to detect integer overflows with a low number of false positives makes these approaches similar to ARBITER. However, these techniques are tailored to a specific class of bugs, while integer overflow is only one example of what ARBITER is designed to detect.

2.2 Binary Analysis: Static vs. Dynamic

Table 2 provides a qualitative comparison of common vulnerability discovery techniques on binary code. Generally, dynamic techniques and static techniques have different focuses: The former sacrifices code coverage (due to the dynamic coverage problem, which leads to a high number of missed bugs) and heavily relies on human effort for harnessing and deployment, while the latter can easily achieve high code coverage at the cost of high false positives. Resource requirements, scaling up, filtering false positives from reported vulnerabilities, and human effort in modeling environment or creating harnesses all directly translate to the essential cost of applying these techniques. This cost contributes to the reluctance of applying these techniques on a wide spectrum of binaries.

Revisiting Table 2 reveals an interesting gap among these vulnerability discovery techniques: The lack of *economic*, *scalable*, and *low-human-inference* techniques with low false positive and false negative rates. Specifically, the absence of such techniques dictates that the use of modern vulnerability discovery techniques must be combined with high cost, in the form of either significant compute or human effort. This motivates our design of ARBITER, a low-cost vulnerability discovery technique on binary code for a set of vulnerability classes that are compliant with certain properties.

3 Vulnerability-Targeted Static Analysis

Our key insight is that, by carefully choosing vulnerabilities with properties that can be leveraged during both static and dynamic analysis, we can properly scope a vulnerability detection approach to combine these two paradigms and achieve high precision while maintaining scalability. To this end, we identified a set of vulnerability "properties" that lend themselves well to static analysis while also providing opportunities for integrating dynamic techniques to improve precision. We identified three such properties:

(P1) Data-flow sensitive vulnerabilities. Vulnerabilities that are *data-flow sensitive* can be discovered by reasoning about data flows between input sources and vulnerability sinks. Note that the scope of data-flow sensitive vulnerabilities is strictly larger than taint-style vulnerabilities, which only includes vulnerabilities that are caused by *missing* sanitization on tainted user input [45]. These are approachable by static techniques, with typical precision limitations, but *also allow for additional dynamic verification of data flows* to increase precision.

(P2) Easily identifiable sources or sinks. During vulnerability discovery, vulnerability sources determine the beginning of data-flow tracking, and vulnerability sinks determine the termination of data-flow tracking. Identifying sources and sinks for many vulnerability classes requires precise aliasing information on an entire binary, which is known to be prohibitively expensive and unscalable. For example, if the source is defined as "any data read from file `/tmp/secret`," identifying all input sources will require correctly analyzing the parameters of every function and syscall that opens a file, and the propagation of file pointers, file descriptors, and relevant data structures. On the contrary, some have sources or sinks that can be identified through examining artifacts that a computationally cheap and scalable analysis generates, such as a CFG. An example of such a sink is "all call sites to the library function `malloc()`."

With this style of sources and sinks, slicing a program from a source to a sink is a well-studied static technique. Though this property is generally considered in the context of static analysis, these slices can be processed by dynamic techniques. This allows for dynamic techniques, such as DSE, to reason about over-approximations in the slicing process, fix such over-approximations, and improve overall system precision.

(P3) Control-flow-determined aliasing. Tracking data-flows almost always involves recovering aliasing information, which requires an expensive, full-binary analysis on the target. However, we observe that the data flows that many vulnerabilities incorporate do not involve pointer dereferencing at all, or when they do, the pointer can always be resolved to one object that is determined by the control flow (*e.g.,* local variables accessed through the stack pointer). We deem such cases of aliasing as control-flow-determined, making this a prerequisite of our targeted vulnerabilities.

Technique	Genre	FP	FN	Vuln. types	Coverage	Resource	Human effort	Scalability	Examples
Fuzzing	dynamic	no	high	crash-introducing	very low	high	harnessing, deployment	high	[1, 2, 7, 10, 11, 30, 37, 50]
Taint tracking	dynamic	no	high	generic	very low	high	harnessing, env., rules	high	[26, 34]
DSE	dynamic	yes	high	generic	low	very high	harnessing, env., rules	low	[34, 38]
DSE-assis. fuzzing	dynamic	no	high	crash-introducing	low	high	harnessing, env.	medium	[5, 6, 16, 17, 27, 41, 49]
Graph-based tech.	static	high	medium	generic	high	low	env., rules, fp	high	[28, 44, 45]
Taint-based tech.	static	high	medium	generic	high	low	env., rules, fp	high	[8, 18, 31, 32]
ARBITER	hybrid	low	low	specific	high	low	rules	high	

Table 2: A qualitative comparison of the strengths and weaknesses of common vulnerability discovery techniques on binary code. "DSE-assis. fuzzing" refers DSE-assisted fuzzing techniques; "Graph-based tech." refers to static analysis techniques that are based on program property graphs; "Taint-based tech." refers to taint-based static analysis techniques. The "high," "medium," and "low" assessment are empirical and illustrative. "Generic" means users can develop violation detection rules for each vulnerability type to discover. "Env." refers to the effort of modeling environment (developing taint propagation rules for library functions in taint tracking, or developing function summaries for library functions in DSE). "FP" refers to false-positive reduction effort, which is mostly done by humans.

From the static side, this property allows ARBITER to adaptively step forward or back on a CFG without having to worry about imprecisions caused by incorrect aliasing information. From the dynamic side, it greatly simplifies aliasing problems resulting from the lack of initialization of the dynamic state.

Choosing the right techniques. The three key properties previously described allow ARBITER to yield a powerful integration of static and dynamic techniques that focus on a subset of common vulnerability classes while retaining sufficient generality to adapt to a range of real-world vulnerabilities. Specifically, we use the static techniques of data flow recovery and program slicing (more details in Section 5). Our choice of dynamic techniques is informed by the vulnerability properties. `P1` defines the scope of vulnerabilities that ARBITER can support and suggests the use of DSE, which can reason about complex value relationships in data flows. `P2` defines a pre-condition to achieve high scalability by providing an opportunity to execute *small slices* of programs. This poses a challenge: These slices typically lack state information necessary for a dynamic analysis. Luckily, `P3`, and the resulting ability to ignore much of the aliasing problem, allows us to effectively apply a DSE technique known as Under-Constrained Symbolic Execution (UCSE). Without `P3`, ARBITER would need to either compute aliasing during static analysis (which is an undecidable problem) or accept high false positives caused by conservative aliasing. Our dynamic techniques are discussed in Section 6.

Thus, ARBITER can reason about vulnerabilities that comply with these properties in both static and dynamic-symbolic contexts. We call these vulnerabilities *Property-Compliant (PC) Vulnerabilities*. To give the reader a better perspective on the concrete scope of our approach, we represent candidate Common Weakness Enumeration (CWE) entries which contain PC-type vulnerabilities in Table 3, along with example CVE entries and suggested sources and sinks. In the remainder of this paper, we will focus on CWEs: 131, 134, 252 and 337: "Incorrect Calculation of Buffer Size," "Uncontrolled Format String," "Unchecked Return Value," and "Predictable Seed in Pseudo-Random Number Generator."

3.1 CWE-131: Allocation Site Overflows

Integer overflows at allocation sites are a serious class of vulnerabilities that can provide attackers with powerful primitives. Such vulnerabilities can cause programs to allocate blocks of memory smaller than the amount of data they are supposed to hold. When data is copied into this memory, the resulting out-of-bound memory accesses can potentially be exploited by attackers to gain code execution.

Occurrence. We observed this as a common pattern in modern software where a custom wrapper function is invoked to allocate memory using common `libc` functions such as `malloc`, `realloc`, or `calloc`. The wrapper function usually requires an argument that denotes the number of bytes to be allocated. Before invoking functions to allocate the requested number of bytes, that number might be increased to accommodate application-specific metadata. When the requested size is very large, this increase can lead to an integer overflow, causing a too-small allocation to be requested.

Static sources and sinks. Naturally, the sinks for this CWE are allocation functions (`malloc`, `calloc`, `realloc`). The sources, in turn, are arguments to the wrapper function that *calls* the identified sinks.

Static data flow. The data flow specification for this vulnerability is simple: we can statically recover all flows from the sources to the sinks, and then reason about them symbolically.

Dynamic symbolic constraints. Our dynamic constraints for CWE-131 ensure that allocation size calculations do not overflow in wrapper functions. The constraints, expressed mathematically here (but procedurally in ARBITER), are as follows:

Let $uint : e \rightarrow uint$ denote the evaluation of an expression as an unsigned integer value. Given an arithmetic expression e passed as argument to a memory allocation function f, and its individual terms $\{e_1, \cdots, e_n\}$. A vulnerability exists iff $\exists e_i \in \{e_1, \cdots, e_n\} | uint(e_i) > uint(e)$.

This constraint is based on the assumption that a function that contains the allocation site never decreases the required length of the memory block[3].

[3] Interestingly, in our large-scale evaluation, we found that this assumption does not always hold, which leads to false positive detections of vulnerabilities. We discuss these cases, and their impact on our results, in Section 8.1.1.

CWE-ID	Description	P1	P2	P3	Source	Sink	CVE
CWE-78	Improper Neutralization of Special Elements used in an OS Command ("OS Command Injection")	✓	✓	✓	1st arg. of `sprintf()`	1st arg. of `system()`	CVE-2016-1334
CWE-190	Integer Overflow or Wraparound	✓	✓	✓	arithmetic op. on return values of `sizeof()`	`size_t` arg. of *	CVE-2017-1000121
CWE-131	Incorrect Calculation of Buffer Size	✓	✓	✓	arithmetic operations	arg. of `malloc()`	CVE-2018-18311
CWE-134	Controlled Format String	✓	✓	✓	*	`printf()` and alike	CVE-2012-0809
CWE-252	Unchecked Return Value	✓	✓	✓	return values of `setuid()`	*	CVE-2013-4559
CWE-337	Predictable Seed in Pseudo-Random Number Generator	✓	✓	✓	return values of `time()`	arg. to `srand()`	CVE-2020-13784
CWE-676	Use of Potentially Dangerous Function	✓	✓	✓	*	1st arg. of `strcpy()`	CVE-2011-0712
CWE-120	Buffer Copy without Checking Size of Input ("Classic Buffer Overflow")	✓	✓	✓	2nd arg. of `read()`	1st arg. of `memcpy()`	CVE-2003-0595

Table 3: Example candidate vulnerability classes that ARBITER supports with sample sources and sinks. The green background indicates a deterministically matched property. The yellow background indicates a conditionally matched property. `P1` refers to data-flow vulnerabilities, `P2` refers to easily identifiable sources and sinks and `P3` refers to control-flow-determined aliasing. In this paper, we specifically explore the automatic detection CWE's 131, 134, 252 and 337.

3.2 CWE-252: Unchecked Return Values

Vulnerabilities can occur if a program does not properly handle an error. These vulnerabilities can provide an attacker with a range of capabilities from Denial-Of-Service to Privilege Escalation [21]. A typical attack vector is a vulnerability in a program that tries to drop privileges to an unprivileged user after a privileged initialization phase. If the return value from the privilege reduction (i.e., `setuid`) is not checked, the program may be vulnerable.

Occurrence. These flaws can occur whenever developers omit return value checks for code that *can*, but almost never *does*, fail. Aside from `setuid` and similar functions, this includes frequently used functionality such as `mmap`, `open`, `close`, and so on.

Static sources and sinks. We decided to focus on system calls that have clear security implications, resulting in the following list of sources: `access`, `chdir`, `chown`, `chroot`, `mmap`, `prctl`, `setgid`, `setsid`, `setpgid`, `setreuid`, `setregid`, `setresuid`, `setresgid`, `setrlimit`, and `stat`. This list is trivially expandable. For sinks, we used the return block of the function (in the binary being analyzed) that is calling the system call's corresponding `libc` API.

Static data flow. To catch cases where the return value was discarded (a common cause of this vulnerability), the data flow specification must capture all flows of the API return value from the source, including those where the return value of the caller did *not* depend on the system call return value.

Dynamic symbolic constraints. The return value of our chosen APIs can either be -1 for error, or something else, for success. ARBITER's ability to reason symbolically allows us to craft a powerful vulnerability constraint for this application: If, at the end of the function, DSE identifies states where the return value from the API call can be *either* success or failure, then there was no check to constrain the value, and an unchecked return value vulnerability exists.

3.3 CWE-134: Uncontrolled Format String

Formatting functions, such as `printf()`, determine the number of arguments at runtime based on a *format string*. This format string, as a required function argument, contains both ordinary characters (printed verbatim) and *format directives*, which specify arguments to be printed, copied, or modified. If a user can control the format directives, they can often use this capability to disclose or overwrite memory values and compromise the security of the program.

Occurrence. These flaws can potentially occur whenever a format string is not constant. This can be the result of lazy programming practices (i.e., `printf(name);` instead of `printf("%s", name);`) or of insufficient filtering in the dynamic construction of format strings. The former frequently occurs in lesser-exercised paths (especially error logging).

Static sources and sinks. We used the set of `libc` functions that would be affected by format string vulnerabilities as sinks. These are `printf`, `fprintf`, `asprintf`, `dprintf`, `sprintf`, `snprintf`, `vasprintf`, `vfprintf`, `vsprintf`, and `vsnprintf`. The sources, in turn, are arguments to the wrapper function that *calls* the identified sinks.

Static data flow. The data flow specification for this vulnerability is simple: we can statically recover all flows from the sources to the sinks, and then reason about them symbolically.

Dynamic symbolic constraints. The dynamic symbolic constraints for this vulnerability check whether the format string used in a sink is a constant or not. If this format string is not a constant, this is flagged as a potential bug.

3.4 CWE-337: Predictable Seed in PRNG

The security of a Pseudo-Random Number Generator (PRNG) depends on the unpredictability of its seed. If an attacker can predict this seed then, regardless of the quality of the PRNG, they can determine the random values used by the program. If these random values are used for security-critical operations, the attacker might be able to violate security.

Occurrence. Unfortunately, a common practice, is to use the current time (in seconds) to seed the PRNG. This makes the random seed predictable, and is responsible for a number of real-world vulnerabilities. Similar common anti-patterns involve the use of process IDs as seeds. These IDs are predictably increasing, also making them poor choices of seeds.
Static data flow. The data flow specification for this vulnerability is simple: we can statically recover all flows from the sources to the sinks, and then reason about them symbolically. While previous vulnerability classes used constant values as negatives, they are treated as positives here.
Dynamic symbolic constraints. DSE is used to confirm the results of the data flow analysis and establish a data flow between the sources and the sinks. There are no additional constraints that need to be applied.

4 ARBITER Analysis Framework

The properties identified in Section 3 enable the effective combination of static and dynamic approaches into ARBITER. In this section, we provide an overview of the ARBITER approach, and discuss, at a high level, how it detects PC vulnerabilities in binary programs.

Figure 1 shows an overview of ARBITER. At a high level, ARBITER combines scalable static analysis techniques with precise DSE techniques. The former allows it to identify a superset of PC vulnerability candidates in the program being analyzed, while the latter acts as a filter for false positives, configurable between precision and scalability. ARBITER functions fully on binary code, and, while its underpinning concepts are well-studied in source code, their use on binary code required several innovative analysis improvements.

Input: The Binary. To analyze binary code, ARBITER needs the binary in question. Unlike purely dynamic analyses (such as fuzzing), the provided binary does not need to be runnable.
Input: Vulnerability Description. As discussed in Section 3, PC vulnerabilities have certain properties that can be used to detect them with static analysis. These properties are provided to ARBITER as a *Vulnerability Description* (VD). A VD is a programmatic representation of the static and symbolic artifact descriptions in Section 3 (e.g., Section 3.1 for CWE-131 and Section 3.2 for CWE-252). Other PC vulnerability classes can be added through the creation of a VD.
Identifying vulnerable flows. ARBITER uses a combination of techniques to identify flows potentially satisfying the VD in the binary. It first searches for VD subjects on a recovered Control Flow Graph, then queries a computed Data Dependency Graph to identify data flows between these subjects matching the provided VD. ARBITER computes paths representing these flows, and promotes these to the next step. This process is described in Section 5.
Verifying vulnerable conditions. ARBITER uses Under-Constrained Symbolic Execution (UCSE) to execute the provided paths and recover symbolic data relationships between the source and sink. If this relationship satisfies the constraints described in the provided VD, the path is promoted to the next step. This process is described in Section 6.
Reducing context-based false positives. ARBITER limits the context sensitivity of its static analysis to achieve scalability. As a result, the data flow detected for a Vulnerability Candidate might be missing constraints that would be present with a higher context sensitivity, resulting in a false positive detection. To alleviate this problem, ARBITER computes a slice from the detected sink with a higher context sensitivity, and symbolically executes it to identify missing constraints. By increasing the context sensitivity level in this step, ARBITER trades scalability for precision. If ARBITER cannot identify any constraints to invalidate an extant Vulnerability Candidate, that candidate will be reported to the analyst as an alert. This process is described in Section 7.

5 Identifying Vulnerable Flows

Taking a CFG and a VD as input, ARBITER builds and queries a data dependency graph (DDG) with respect to the vulnerability sources/sinks specified in the VD. The resulting candidate vulnerable data flows will be verified by the dynamic analysis component of ARBITER. This section will focus on the unique static data flow tracking technique that drives ARBITER, how the DDG is created, and how vulnerable data flow candidates are identified.

5.1 Precise Static Data Flow Tracking

Graph-based and taint-based static analyses were used to find taint-style vulnerabilities in both source code and binary code. While PC vulnerabilities are similar, reasoning about them may require more precision during data-flow tracking. ARBITER does this with a novel static data flow tracking technique, named `SymDFT`, that focuses on *precision* and *scalability* while sacrificing soundness. Given any starting point (usually the beginning of a function), `SymDFT` statically emulates the basic blocks in a context-sensitive and path-sensitive manner, and models registers, memory (as a flat memory model that covers global regions, stack, and heap), syscalls, and accesses to and from file descriptors (this captures accesses to file and network sockets). `SymDFT` achieves high precision of modeled data by using a symbolic domain, i.e., tracking unknown variables and symbolic expressions during analysis. Next we detail the key design decisions of `SymDFT`.

Traversal policy. On a function level, `SymDFT` traverses the basic blocks inside the function in topological order, which ensures a block is always visited after all its predecessors are visited. Once a call to a callee is encountered, `SymDFT` will analyze the callee function and use the returned abstract state to proceed the analysis at the return site.

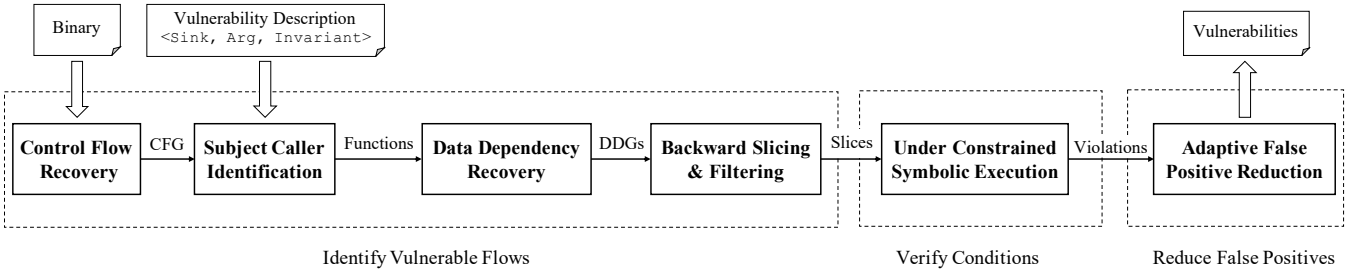

Figure 1: ARBITER's workflow. ARBITER discovers vulnerabilities in three steps: identifying vulnerable flows, verifying conditions, and reducing false positives.

Branching policy. `SymDFT` incorporates an idea underpinning forced execution and blanket execution [12]: At each branching point it forks the abstract state and follows *all* branches, regardless of branch feasibility: `SymDFT` collects symbolic path predicates (i.e., symbolic constraints) as the emulation progresses, but it does not evaluate the satisfiability of these constraints. While this sacrifices precision, ARBITER restores precision in the verification step described in Section 6.

State merging policy. Abstract states at the same binary address are merged immediately, where each state is assigned a *merging label*, and the concrete and symbolic expressions are merged into a form of an if-then-else expression with the merging labels being the conditions.

Termination. To ensure analysis termination, `SymDFT` employs an aggressive termination strategy: (1) When the call depth exceeds 2 functions, `SymDFT` forges an abstract return value instead of analyzing the callee and (2) Each loop is visited at most 3 times. This aggressive termination strategy trades a decrease in soundness and precision for scalability.

5.2 Building DDGs and Extracting Flows

ARBITER first identifies sources or sinks on the CFG and recognizes the relevant functions. Then ARBITER builds a DDG for the function containing each source or sink. Note that these DDGs do not include any of the callers, which will be further discussed in Section 7. Knowing sources and sinks, identifying all vulnerable data flows on a DDG is as trivial as traversing from sources or sinks and generate subgraphs.

When only sinks are described in the VD, ARBITER builds a DDG for each function (and callees) where sinks are found, instead of building a full-program DDG covering all potential input sources. This significantly improves scalability of DDG generation and vulnerable flow identification at the cost of precision, which ARBITER corrects in the verification step.

6 Verifying Vulnerable Flows

Candidate vulnerability data flows collected during static analysis are not necessarily vulnerable because (1) they are unsatisfiable or (2) the path predicates make the vulnerability untriggerable even if the flow is valid. ARBITER uses under-constrained symbolic execution to verify each candidate data flow and eliminate those unsatisfiable or not vulnerable.

6.1 Under-Constrained Symbolic Execution

Due to incomplete CFGs and the dynamic coverage problem, ARBITER usually cannot derive a feasible execution path that starts from the entry point of the target binary and reaches the vulnerability sink. Instead, binary program slices that candidate data flows correspond to usually do not include code that is necessary for program state initialization. In source-based analyses, *under-constrained symbolic execution* (UCSE) is a viable approach to performing DSE on incomplete program states [13, 29]. UCSE allows symbolic execution of arbitrary functions without initializing data structures or modeling environments. All values that are not available in the incomplete state (e.g., missing parameters and unknown return values from callees) are deemed as *under-constrained* variables. When an under-constrained variable is used as a pointer, UCSE will initialize the variable as a pointer that points to a freshly allocated memory region with the same size the pointer type specifies. If the variable is not used as a pointer, an under-constrained variable will be more constrained with symbolic constraints applied to it.

In ARBITER, we extend UCSE to support binary code. UCSE fundamentally suffers from the aliasing problem by assuming under-constrained pointers do not point to any existing objects (or in binaries, any allocated regions in memory). Luckily, `P3` in PC vulnerabilities dictates that no aliasing relationships that are uncaught by the control flow exists, which sidesteps the aliasing problem in UCSE.

Two challenges arise because type information is unavailable on stripped binaries. We present them and our solutions.

Shadow memory allocation. Due to lack of type information, UCSE does not know the size of the memory region to allocate. In ARBITER, UCSE does not determine the sizes of the memory regions that are allocated for under-constrained pointers. Instead, we assume each under-constrained pointer points to a *shadow* memory region that is completely isolated from the stack, the heap, and other shadow memory regions.

All pointers derived from an under-constrained pointer can only be used to access the same shadow region.

Adding necessary constraints. UCSE on source code relies on information about sizes of variables for constraining pointers (to within the variable) and finding violations (when a pointer access goes beyond the range of the variable it should point to). Such information is unavailable on binary code, making constraining pointers and variables difficult, and certain memory accesses caused by under-constrained pointers or variables may corrupt critical data structures on the stack or in the global region (e.g., a fully constrained pointer, a stored return address, etc.), which will lead to unsound symbolic states that do not exist during execution. Hence, our UCSE aggressively adds constraints to variables and pointers so that key data on the stack or in the global region cannot be overwritten. Note that this way, our UCSE will not find vulnerabilities (e.g., stack buffer overflows) that involve overwriting these critical data structures, but such vulnerabilities are not within PC and are thus outside the scope.

6.2 Collecting Data Flow Constraints

For each candidate vulnerability data flow, ARBITER runs UCSE on a program slice that is generated by traversing the CFG from each sink. At the sink, ARBITER extracts the symbolic expressions that represent data dependencies between the source and sink and collects path predicates (constraints). If the constraints corresponding to the VD are satisfiable, ARBITER reports a potential VD violation.

7 Adaptive False Positive Reduction

Despite the satisfiability filtering of candidate vulnerable data flows in the UCSE step, the lack of context still leaves a large number of false positives. Because ARBITER works inside a function, it does not have context-sensitive information that might influence the control- and data-flow of a function.

We filter out false positives by increasing the analysis context *once a vulnerability is detected in a smaller context*. ARBITER recursively identifies all call sites of the current function and proceeds to analyze in the context of the calling frames. ARBITER recursively continues to expand the context caller by caller until a predefined recursion limit is reached.

The recursion limit can be chosen by the analyst adaptively based on the number of reports and time constraints, as each additional context could lead to an exponential increase in analysis time. Our results show that each context expansion steps cuts alerts by roughly a factor of two (shown in Section 8.1), and three reduction steps reduce false positives sufficiently for results to be manually verified.

Class	Alerts	True Positive	False Positive		UT
			VFP	DFP	
CWE-131	436	194	184	11	47
CWE-134	158	12	139	3	4
CWE-252	159	83	15	56	5
CWE-337	377	372	0	2	4

Table 4: Summarizing the results of the different experiments. "VFP" refers to the number of false positives classified as Vulnerability False Positive. "DFP" refers to the number of false positives classified as Description False Positive. "UT" refers to the number of reports that we could not triage due to software complexity.

8 Evaluation

We performed several experiments to measure the effectiveness of ARBITER in terms of both its performance and its ability to identify bugs in real-world software, on a large scale, while maintaining a high precision. The evaluation was performed on a Kubernetes cluster of Intel Xeon CPUs at 2.30GHz, with one core and 4GB of RAM dedicated to the analysis of each binary. The prototype of ARBITER was implemented atop `angr` [38].

We discuss three separate sets of experiments: a performance evaluation on the `coreutils` Linux binaries to measure the computational cost of different parts of our techniques (presented in Appendix C), a comparative experiment on the Juliet program analysis test suite and several other Linux programs to position ARBITER with respect to state-of-the-art, and a large-scale evaluation on the Ubuntu 18.04 software repository (containing 76,516 binaries) to assess ARBITER's ability to detect vulnerabilities in the CWE-131, CWE-252, CWE-337, and CWE-134 vulnerability classes.

8.1 Effectiveness Evaluation

To measure ARBITER's ability to find bugs in real code, we performed a large-scale effectiveness evaluation on all userspace binaries in the x86-64 Ubuntu Linux 18.04 repositories. Out of the 1,852,152 ELF files in the Ubuntu Linux 18.04 repositories, we excluded 1,708,122 kernel modules (on which ARBITER is not designed to function), and 64,101 symbol-only ELF files (containing debug symbols for other ELFs)[4]. The resulting 79,929 ELFs contained 79,343 unique files. The underlying `angr` framework failed to process 2,829 of these, leaving us with a final dataset size of 76,516.

We checked each binary against ARBITER's four implemented VDs: CWE-131, CWE-252, CWE-134, and CWE-337. Across the four vulnerability classes, with a recursion limit of three for the Adaptive False Positive Reduction step, ARBITER raised 1095 alarms. We manually analyzed all of these 1095 alarms and were able to classify 1036 alarms into 636 True Positives and 400 False Positives. We differentiated the False Positives between false positives stemming

[4] In our dataset of nearly 65k packages, 600 packages have on average 4,000 kernel object files each.

from insufficient precision in the analysis (which we term a *Vulnerability False Positive*) and false positives due to imprecision the VD itself (a *Description False Positive*)[5]. The 410 False Positives included 338 Vulnerability False Positives and 72 Description False Positives. For the remaining alerts, we could not classify them into True or False Positives due to our unfamiliarity with the applications and their complexity.

Table 4 presents a summary of ARBITER's alerts. For the curious reader, we include detailed information about the numbers of binaries, functions, sinks, paths, and states that ARBITER dealt with through various stages of its analysis on the different vulnerability types in Appendix D.

We are responsibly disclosing vulnerabilities to the respective developers. Appendix A lists public (addressed or WONTFIX) vulnerability disclosures. Some vulnerabilities have now been fixed due to our efforts (e.g., CVE-2018-18311), while others were vulnerabilities rediscovered by ARBITER but not yet deployed to Ubuntu repositories (e.g., CVE-2016-5636, CVE-2017-1000158, and CVE-2020-7105).

Case Studies. We present several case studies of ARBITER's findings in Appendix E. One case study is especially interesting involving OCaml.

During our triaging of ARBITER's CWE-131 alerts, we found 26 reports where the buggy function was almost identical. When we checked the source code, we found that the programs were all written in OCaml, and *the OCaml source code of the functions in question did not contain any buggy behavior*. Upon further investigation, we found that all 26 alerts were caused by a bug introduced *during compilation by the OCaml compiler*. We investigated this thoroughly and developed a Proof of Concept that triggered an integer overflow and subsequently an out-of-bound heap access that eventually crashes the program. This bug affects *all 32-bit OCaml programs* compiled with affected releases (up to the latest one) of the OCaml compiler. We disclosed to the OCaml developers, and they are currently investigating the issue.

8.1.1 CWE-131: Overflows at Allocation Sites

ARBITER raised 436 CWE-131 (Incorrect Calculation of Buffer Size, described in Section 3.1) alerts in 444 functions across 366 binaries on our dataset. Through manual static analysis, we determined that 194 of these were True Positives, 184 were Vulnerability False Positives, and 11 were Description False Positives. We were unable to triage 47 alerts across 34 binaries due to the complexity of the vulnerable code and our lack of familiarity with the code bases in question.

Description False Positives. 11 alerts reported across 10 binaries show that the predicate we used in our CWE-131 VD does not always imply the presence of a bug. Our VD goal was to model an integer overflow vulnerability in the argument to an allocation invocation. However, the predicate in the VD is based on a typical pattern for an integer overflow, not a specific integer overflow marker. Specifically, the pattern describes the situation of a size variable being passed as an argument to an allocation wrapper, and overflowing during computation, resulting in a smaller variable being passed to the allocator. In 11 out of 150,077,458 functions in the Ubuntu dataset this does not hold: a computation performed by the wrapper function legitimately reduces the allocation size, without undergoing an integer overflow.

Vulnerability False Positives. We identified several major reasons for VFPs among the CWE-131 alerts.

Unrealistic Execution Requirements. Triggering some detections may require executing callbacks an unreasonable number of times. For example, a detection that depends on a global counter becoming very large might need the function that increments that global counter to be executed billions of times, and is thus not triggerable in practice.

Unrealistic Data Requirements. Some alerts are generated for functions that *would* suffer an integer overflow when provided with a string of, e.g., 2^{64} bytes. As such data is infeasible to produce or transfer, these bugs are not triggerable. By investigating these cases further, we found that ARBITER raised alerts in 24 functions across 24 binaries because it assumed that 2^{64}-1 byte-long strings were practical to have in memory.

Dead code. Some detections that appeared to be practical bugs were in dead code that could not be triggered.

Insufficient analysis context. In some cases, caller functions higher up in the call stack introduce constraints that make the vulnerability impossible to trigger in practice, but ARBITER does not analyze to a high enough caller level to reason about those constraints. For example, the `alttab` binary uses LibPNG to load a PNG, and contains a 4-byte integer overflow caused by image dimensions. Although the PNG specification allows for PNG height and width up to 2^{32}, LibPNG limits them to 2^{16}, making the bug untriggerable.

Signedness side-effects. In situations where an overflowed integer is sign-extended before being used in an allocation function, the extended value becomes huge, and the process exits because the allocation fails.

8.1.2 CWE-252: Unchecked Return Values

In this template, ARBITER symbolically executes paths starting from the sink and tries to reach the end of the function that contains the sink. Then, during FP reduction, it executes paths from the outermost caller to the sink and continues execution until the end of that caller. This differs from the other three templates where ARBITER only explores from the caller up until the sink because, in CWE-252, the otherwise-unchecked return value could later be checked by a caller function.

For security-critical APIs listed in Section 3.2, ARBITER produced a total of 159 CWE-252 alerts in 135 functions

[5] As the VD we used for both experiments were proven incorrect in at least a few cases, we feel it is important to discuss both types of false positives separately.

across 126 binaries. We manually triaged these to identify 83 True Positives, 139 Vulnerability False Positives, 3 Description False Positives, and 4 alerts that we could not triage.

Description False Positives. Of the filtered list of 159 reports, we found 53 reports that violated the assumptions we had made while generating our VD for several reasons.

Atomic operation confusion. There were two detections where the subject's return value was overwritten with a default value using a single instruction *cmov*, which is a conditional mov instruction. ARBITER assumed that if a single state can satisfy either constraint for the return value of the subject, it must be reported as a bug. However, in these cases, the *cmov* instruction did not generate a new state but added a constraint on the return value of the subject. Our analysis incorrectly classifies these instances as bugs. Similarly, two other false positives were caused by *setz*. We can solve this problem by modifying the underlying code lifting mechanism to split a single *cmov/setz* instruction into two branches.

Unsupported error values. We had assumed that the return value of the subject could be either SIZE_MAX or UINT_MAX to return errors as 64- or 32-bit values respectively. Therefore, our VD would check if the value returned is either SIZE_MAX or UINT_MAX. All sinks VD except for *mmap* return UINT_MAX on error, while *mmap* returns SIZE_MAX on error. In the case of *mmap*, both UINT_MAX and 0 are considered successful return values. Therefore, our analysis incorrectly labels these cases as True Positives. We found 52 cases where the return value of *mmap* was compared against UINT_MAX and was (incorrectly) reported.

Vulnerability False Positives. In this case, false positives are usually caused by limitations in ARBITER's analysis engine.

Duplicate sinks. When a function has more than one sink, ARBITER would create symbolic expressions for the return value of each sink. However, the dependency between the sink and the generated symbolic expression is not preserved. Therefore, when the return value of one of these sinks is checked for error and the other is not, the constraints applied on the unchecked return value would be satisfied. ARBITER incorrectly classifies the first sink as the alert location when the actual alert is linked to the second sink. However, it should be noted that ARBITER also correctly identifies the alert linked to the second sink. We found eight instances of such alerts.

Unsupported API functions. When some functions return an error code, the program will call function *__errno_location* to get the address of the *errno* variable for the current thread. The program will then dereference the returned address to retrieve the error code. When *__errno_location* is invoked, the address it returns is stored in register `rax`. But, angr does not overwrite or clear `rax` after simulating *__errno_location*. This causes ARBITER to assume that the return value of the sink (instead of the address of *errno*) is being dereferenced, and according to the VD for CWE-252, the dereferenced value must also be checked. ARBITER raises an alert when the check is missing. We found six such alerts.

Ignoring error checking functions. Recall that for this VD, in the FP reduction step, ARBITER would symbolically explore paths from a caller function until the sink after which it would continue exploring paths that lead to the end of the caller function. This is different from other VD's where ARBITER only symbolically explores paths from a caller function up until the sink. The reason is that other VD's can determine the existence of a bug at the sink, unlike this VD.

This increase in complexity of the paths explored leads to a large number of timeouts with each additional level of FP reduction. Therefore, we used a feature of the underlying framework that does not symbolically explore calls to any function after the sink has been invoked. This can lead to incorrect results if the return value of the sink is passed to a sub-routine as an argument. If this sub-routine is tasked with checking the value of its argument, ARBITER will not detect this check because the sub-routine will not be explored at all.

We found only one case where such a sub-routine was used to check the return value of the sink.

8.1.3 CWE-337: Predictable Seed in PRNG

ARBITER reported 377 CWE-337 alerts in 372 functions across 350 binaries. Through manual triaging, we identified 372 True Positives, 2 Description False Positives, and 4 cases that were too complex to triage.

Description False Positives. These 2 Description False Positives occurred because a seed used in the call to *srand* was generated by combining (e.g., with an `xor` operation) the return value of *time* with a value read from the kernel's random number generator (using */dev/urandom*). Because our description did not test if anything *other* than the time was used to seed the PRNG, ARBITER had no way to filter out these cases.

Vulnerability False Positives. We did not identify a Vulnerability False Positives in the results. The common case was `srand(time(NULL));`.

8.1.4 CWE-134: Uncontrolled Format String

ARBITER raised 158 CWE-134 (Controlled Format String, described in Section 3.3) alerts across 139 functions in 119 binaries. Through manual triage, we determined that 12 of these reports where True Positives, where the user could control the format string and trigger a format string vulnerability. However, we found that 142 of these reports did not lead to a format string vulnerability and therefore are false positives, mostly (as discussed next) due to insufficient context information even with 3 levels of FP reduction.

Description False Positives. Among the 142 false positive reports generated by ARBITER during this experiment, three were caused due to a violation of the assumptions we made while generating our VD. Our objective was to find invocations of the `printf` family of functions where the format specifier was not a constant string.

However, our VD would flag situations where a string is generated by combining multiple constant strings using `sprintf`-like functions. We found 1 situation where `sprintf`-like were invoked to write constant strings into a stack buffer. This stack buffer was then used as the format specifier for a different sink. Because the stack buffer is not a constant value, ARBITER flagged such situations as positives.

In another two cases, the format specifier was set to NULL. Since this value is not considered as a constant string, these two reports were alerted by ARBITER.

Vulnerability False Positives. 36 false positives were caused by functions similar to `G_gettext` and `__dcgettext`. These functions translate a text string into the user's native language. Because ARBITER lacked this knowledge, it assumed that the string returned by these functions was not constant.

A majority of the false positives occurred due to insufficient context information. We found that 103 of the false positive reports occurred since a caller that was 4 levels (or more) up the callstack invoked the subsequent functions with a constant string as the argument. Since our false positive reduction step only included three level callers, it could not detect these constants and misclassified them as positives.

8.2 Comparative Evaluation

To understand how ARBITER fits into the current state of the art, we perform a comparative evaluation across several experiments. We demonstrate the advantage of ARBITER's eschewing of traditional dynamic analysis by evaluating against AFL, compare ARBITER against static analysis techniques that have the advantage of source code access but lack ARBITER's symbolic execution capabilities (CodeQL and Infer), and use ARBITER to analyze the Juliet static analysis test suite and identify previously unreported bugs.

8.2.1 ARBITER vs. Dynamic Analysis (AFL)

While the approaches and techniques we used to detect vulnerabilities are different from the dynamic approach of fuzzing, a comparative evaluation of a popular fuzzer can demonstrate strengths that ARBITER provides over fuzzing.

We generated POCs for 25 integer overflow vulnerabilities in 15 binaries that ARBITER had reported, and evaluated whether AFL is able to find these vulnerabilities. Of these 15 binaries that contain vulnerabilities, only seven are standalone binaries, with the remainder being shared objects that required a harness to execute. Two of these seven binaries could not be run without additional manual environmental configuration, such as GUI/network. For the remaining five, we used a wrapper that would use the bytes generated by AFL as the relevant command-line arguments or environment inputs in order to fuzz them. Because, while creating the POCs for ARBITER's detection, we had already understood the appropriate command-line argument/environment variable that triggers the crash, we modified the wrapper to fuzz only that particular command-line argument/environment variable value.

We ran AFL for 24 hours on one core per binary. In two, AFL found the same bug as ARBITER. In one, AFL found a different crash bug than ARBITER, in a different vulnerability class. In the remaining two, AFL did not find a bug.

When we tried fuzzing these binaries *without* explicitly controlling the argument/environment variable that we identified with ARBITER's help, AFL could not trigger the crash after 24 hours of fuzzing each binary.

These results suggest that a combination of fuzzing and the techniques proposed in ARBITER might be a promising area of research, though recent trends in hybrid fuzzing research make that a fairly predictable statement. They also show that ARBITER *does* find bugs that AFL misses, and does so without the need for manual harnessing.

8.2.2 ARBITER vs CodeQL

CodeQL [35] is a source code analysis engine available on the LGTM [36] platform. We wanted to evaluate whether CodeQL could detect any of the errors we identified with ARBITER. The queries that match our two implementations of ARBITER the closest are *Inconsistent operation on return value* and *Overflow in uncontrolled allocation size*. Note that CodeQL's only analysis is in a semantic domain on a source code level; hence, the queries are fundamentally different from ARBITER's symbolic domain.

We found 11 programs with CWE-252 ARBITER alerts and 54 programs with CWE-131 ARBITER alerts on LGTM. LGTM identified no inconsistent return value operations and only one true positive allocation size overflow alert on these programs. However, it did raise 6 alerts that we matched those that we manually determined to be false positives and 3 alerts that matched those we could not effectively triage.

These results show that ARBITER can identify bugs missed even by source-level analyzers such as CodeQL.

8.2.3 ARBITER vs Infer

Infer is a static program analyzer for Java, C, and Objective-C, written in OCaml [4, 14]. Internally, Infer uses separation logic and bi-abduction to reason about bug classes such as null pointer dereferences, memory leaks, coding conventions, and unavailable API's. Unfortunately, in 2017 Infer removed the support for the bug class of ignored return values [15]. Nevertheless, Infer still supports the detection of integer overflows in Java, C, and Objective-C programs.

We evaluated Infer on a subset of the integer overflows for which we created POCs. Four targets have build system unsupported by Infer, so we ran the tool manually on the particular source code files. Infer was unable to detect integer overflows in these targets. We believe that Infer's separation logic analysis capabilities are less powerful when it comes

Listing 1 An example of Juliet v1.3 CWE190 faulty test cases.

```
int goodB2GSink(unsigned int data) {
  ...
  if (abs((long)data) < (long)sqrt((double)UINT_MAX)) {
    unsigned int result = data * data;
    printUnsignedLine(result);
  }
  ...
}
```

to sink-triggered vulnerabilities, only enabling detection of shallow integer overflow vulnerabilities. This again shows the strength of ARBITER's combination of static analysis and symbolic execution.

8.2.4 ARBITER on Juliet

We evaluated ARBITER's integer overflow VD against test-cases in the Juliet Test Suite (v1.3), a synthetically generated data set created to assess the capabilities of static analysis tools. Version 1.3 contains many test cases written in C/C++ for multiple CWE types. We adapted our CWE-131 vulnerability description CWEs, CWE-680 (Integer Overflow to Buffer Overflow), and CWE-190 (Integer Overflow), and expanded it to C++ by adding `new` to our list of sinks. This represents a total of 15,680 test cases, of which 4,536 have intentional bugs described by the Juliet documentation.

Description False *Negatives*. Several test cases (for CWE-680) initialize variables to constant values that are too large to be used safely in an arithmetic operation and can lead to integer overflows. However, ARBITER's integer overflow vulnerability description does not reason about constant-value sources. Thus, we misclassified 52 test cases as negatives when in fact, Juliet considers them to be bugs.

Signedness adaptation. When reasoning about overflows, we could determine the signedness of the data involved based on the type signature of the sink function (i.e., `malloc` takes unsigned integers). However, all of our prior sinks used unsigned integers, while Juliet has provided several signed sinks.

Juliet's unsigned overflow test cases use the sink `printUnsignedLine`, which takes an unsigned argument, while the signed test cases use the sinks `printIntLine`, `printHexCharLine` and `printLongLongLine`.

For the unsigned test cases, we used the same constraints as the integer overflow experiment because the comparison by default is unsigned. For the signed test cases, we modified the constraints to use a signed comparison.

Unexpected positives. Examining the results, we found that several test cases in the Juliet test suite that were classified as "safe" are, unknowingly to the developers, vulnerable.

A snippet of a faulty test case, performing a square operation on a 32-bit unsigned integer, is shown in Listing 1. The code attempts to ensure that the value is small enough to perform the square operation without an overflow by computing the absolute value of this integer and compares it with an upper bound.

The absolute value is calculated using the `abs` function which accepts a *signed* integer as argument and returns another signed integer. This conversion of an unsigned data type to a signed causes a large value to pass the check and subsequently lead to an overflow. For example, if the value of data is `UINT_MAX` (0xFFFFFFFF), the function `abs` will return the value 1, passing the size check. The result of the multiplication is the value 0xFFFFFFFE00000001, resulting in a 32-bit integer overflow.

ARBITER identified 190 such faulty test cases within the Juliet test suite. After ensuring that they were actually triggerable bugs (by crafting a POC for each), we reported it to the developers, but have not heard back from them. Through the rest of this section, we consider these 190 test cases as true positives in the ground truth.

ARBITER's results. We ran ARBITER with the Adaptive False Positive filtering tuned to different levels: 0 (no additional filtering), 1, and 2. At 0 levels of false positive filtering, ARBITER reported had 2,411 TP and 2,890 FP. At one level, the FP count reduced to 2,121, and the TP count reduced to 2,197 (likely as a result of timing out on symbolic tracing of several paths). At two levels, the TP count increased to 3,234 (because of additional symbolic constraints that increase the tractability of UCSE), and the FP count fell to 1,015. ARBITER achieves a precision of 0.77 and recall of 0.57. We show the data in Figure 2.

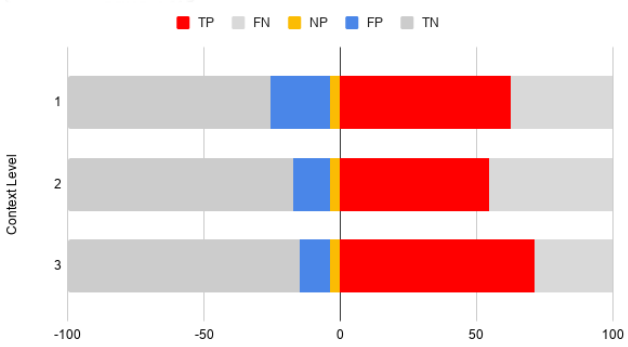

Figure 2: The result of evaluating ARBITER on the Juliet Test Suite v1.3 on CWEs 190 and 680. ARBITER found 3,234 True Positives, 1,015 False Positives, 9,581 True Negatives and 2,580 False Negatives. NP specifies the 190 unexpected "New Positives" that ARBITER found.

Name	Genre	Source needed	Precision	Recall
ARBITER	Hybrid	False	0.77	0.57
CLORIFI [23]	Hybrid	True	1	0.95
Li et al. [24]	Hybrid	True	1	0.973
Ribeiro et al. [33]	Static	True	0.678	0.958
Yang et al. [47]	Static	False	1	0.995

Table 5: A comparison of strengths and weaknesses of tools that evaluate on the Juliet data set.

Literature comparison. There are several tools that evaluate against the Juliet dataset, and we can directly compare our results with theirs. We present this comparison in Table 5.

Interestingly, these tools use a diverse set of approaches. Most of them require source code to function, with source code being a much more robust area of static analysis (but not, as shown in our evaluations against CodeQL and Infer, always better than ARBITER). One that does not, the technique proposed by Yang et al. [47], uses a machine learning model that recognizes vulnerability patterns to detect them and reports a precision of 1.00 and a recall of 0.995. Of course, models trained to recognize vulnerable patterns in synthetic datasets are prone to overfitting: Yang et al. report that the same model trained on the Juliet Test Suite achieves a precision of 0.0036 and a recall of 0.539 when used on real world vulnerabilities from the NVD database.

Importantly, none of these tools detected the faulty test cases that ARBITER detected in the Juliet test case. For source-code-based techniques, this is likely due to lack of modeling for the corner case behavior of the `abs` function, which ARBITER does by functioning on the binary level. For machine-learning techniques, this is likely due to training on Juliet's stated ground truth, which is inaccurate in this case.

9 Limitations and Discussion

ARBITER has a number of analysis limitations as well as limitations on the type of vulnerabilities.

CWE coverage. ARBITER's effectiveness is limited to CWE types that can be classified as PC vulnerabilities as described in Section 3. Some of the limitations of the PC requirement stem from underpinnings of the static analyses used by ARBITER, specifically for quickly finding analysis targets and supporting the limited analysis context. Others are imposed by ARBITER's use of symbolic execution to verify alerts.

An example CWE that does *not* satisfy ARBITER's requirements is CWE-362 (Race-Conditions). The difficulty of identifying sinks (variables shared between threads) CWE-362 violates `P2`, the need to reason about variable accesses across thread contexts violates `P3`, and the impact of execution sequence on the data flow analysis result violates `P1`.

Static analysis limitations. The static analysis must be able to find the VD subjects first. Inlined functions, statically linked, or stripped programs make this task harder but not impossible. Furthermore, the analysis needs to recover the control- and data-flow up to a certain quality. Large or obfuscated programs can make this task harder and very time-consuming, which affects the scalability of ARBITER. Finally, as we saw in the evaluation, ARBITER *does* encounter issues caused by classical static analysis problems, including aliasing, resulting in an increase of false positive rates.

Dynamic analysis limitations. ARBITER's dynamic component must execute program slices using Under-Constrained Symbolic Execution. The limitations of symbolic execution are well known and effective solutions are still being studied. If these slices are too large, the dynamic execution can suffer state explosion, leading to slowdown or failure of this step. Additionally, due to the under-constrained nature of our DSE phase, ARBITER suffers from false positives through the lack of scope and the corresponding constraints on the program slice under test.

Take-away. As our evaluation results show, despite limitations in its underlying techniques, ARBITER can (and *does*) achieve actionable results on large-scale analyses of real-world software. Analyzing 76,516 binaries with purely dynamic analysis is simply not feasible, and, until now, scalable and precise static analysis techniques have not been developed. Despite its limitations, ARBITER takes an important step toward practical, large-scale binary analysis.

VD Implementation. The first challenge in implementing a VD is to understand the vulnerability type that is being targeted. An analyst must match components of the target vulnerability (e.g., as sources and sinks, data-flow properties) with the corresponding properties for the VD in ARBITER. Once this matching is performed, it is then relatively straightforward to implement the VD using ARBITER's API. However, there are some edge cases that may only surface once the VD has been implemented. During our evaluation, we found (and fixed) some issues that caused a number of false positives such as the case of Unrealistic Data Requirements in Section 8.1.1 and the cases of the `gettext` family of functions as described in Section 8.1.4. While new edge cases can arise when implementing a new VD, we have found that, once these edge cases were identified, fixing them was trivial. More details about the specific implementations of existing VDs in ARBITER is presented in Appendix B.

Applying ARBITER. Even though ARBITER's false positive rate of nearly 50% improves on state-of-art techniques that target binary applications with similar scalability [38], prior studies have shown that less than 20% of developers are willing to accept false positive rates greater than 40% [9]. In our experience, ARBITER's results are best combined with manual audits of the target application to verify if it is possible to trigger the vulnerability condition, but these results could be used in combination with existing dynamic approaches that are aimed towards verifying static analysis reports which can eliminate false positives [3, 22].

10 Conclusions

In this paper, we presented ARBITER, a novel framework, combining the strengths of static and dynamic analysis techniques for identifying complex security vulnerabilities in any depth of a program at scale. Using ARBITER, we could verify VDs on a substantial subset of Ubuntu package repositories, identifying and reporting many security vulnerabilities.

We envision that ARBITER will be used in the software development life cycle and by security researchers to protect software by analyzing and verifying VDs. To that end, we have presented an in-depth evaluation of ARBITER, and we are releasing it as an open-source framework.

11 Acknowledgement

The authors would like to thank Jacob Torrey and the anonymous reviewers for their careful feedback along with the opportunity for revision which greatly improved the clarity of this paper. This project has received funding from the Defense Advanced Research Projects Agency (DARPA) under Grant No. HR001118C0060 and FA875019C0003 and The Office of Naval Research under Grant No. N00014-17-1-2897.

References

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[2] Frederic Besler. Circumventing fuzzing roadblocks with compiler transformations, 2016.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[4] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[5] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Laszlo Szekeres, Stephen McCamant, and Dawn Song. Transformation-aware exploit generation using a hi-cfg. Technical report, California Univ Berkeley Dept of Electrical Engineering and Computer Science, 2013.

[6] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015.

[7] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[8] Ronny Chevalier, Stefano Cristalli, Christophe Hauser, Yan Shoshitaishvili, Ruoyu Wang, Christopher Kruegel, Giovanni Vigna, Danilo Bruschi, and Andrea Lanzi. Bootkeeper: Validating software integrity properties on boot firmware images. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 315–325, 2019.

[9] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 332–343, 2016.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[11] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *Proceedings 2021 Network and Distributed System Security Symposium, Virtual*, 2021.

[12] Manuel Egele, Maverick Woo, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, San Diego, CA, 2014. USENIX Association.

[13] Dawson Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 1–4, 2007.

[14] Facebook. Infer. https://fbinfer.com/.

[15] Facebook. [infer][biabduction] disable the reporting of return value ignored · facebook/infer@2a8e192. https://github.com/facebook/infer/commit/2a8e1922801e09180143f1b5a6917f6cab3ffb4e.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[17] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20–27, 2012.

[18] Ivan Gotovchits, Rijnard van Tonder, and David Brumley. Saluki: finding taint-style vulnerabilities with static property checking. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2018.

[19] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[20] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2019.

[21] Sebastian Krahmer. The evilness of setuid(getuid()). https://c-skills.blogspot.com/2008/01/evilness-of-setuidgetuid.html.

[22] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3559–3576, 2021.

[23] Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon, and Heejo Lee. Clorifi: software vulnerability discovery using code clone verification. *Concurrency and Computation: Practice and Experience*, 28(6):1900–1917, 2016.

[24] Hongzhe Li, Jaesang Oh, and Heejo Lee. Detecting violations of security requirements for vulnerability discovery in source code. *IEICE TRANSACTIONS on Information and Systems*, 99(9):2385–2389, 2016.

[25] David Molnar, Xue Cong Li, and David A Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, pages 67–82, 2009.

[26] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[27] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[28] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 2015.

[29] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[30] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[31] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. {BootStomp}: On the security of bootloaders in mobile devices. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 781–798, 2017.

[32] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1544–1561. IEEE, 2020.

[33] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. Ranking warnings from multiple source code static analyzers via ensemble learning. In *Proceedings of the 15th International Symposium on Open Collaboration*, pages 1–10, 2019.

[34] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[35] Semmle. CodeQL. https://securitylab.github.com/tools/codeql.

[36] Semmle. LGTM. https://lgtm.com/.

[37] Kostya Serebryany. libfuzzer–a library for coverage-guided fuzz testing. *LLVM project*, 2015.

[38] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[39] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 473–486, 2015.

[40] Lingyun Situ, Linzhang Wang, Yang Liu, Bing Mao, and Xuandong Li. Vanguard: Detecting missing checks for prognosing potential vulnerabilities. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, pages 1–10, 2018.

[41] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[42] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*, 2009.

[43] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[44] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[45] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812. IEEE, 2015.

[46] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510, 2013.

[47] Mingyue Yang. *Using Machine Learning to Detect Software Vulnerabilities*. PhD thesis, 2020.

[48] Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. Statsym: vulnerable path discovery through statistics-guided symbolic execution. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–120. IEEE, 2017.

[49] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[50] Michal Zalewski. american fuzzy lop, 2014. https://lcamtuf.coredump.cx/afl/.

[51] Yang Zhang, Xiaoshan Sun, Yi Deng, Liang Cheng, Shuke Zeng, Yu Fu, and Dengguo Feng. Improving accuracy of static integer overflow detection in binary. In *International*

Symposium on Recent Advances in Intrusion Detection, pages 247–269. Springer, 2015.

A Disclosures of Vulnerabilities Found by ARBITER

The following is a list of vulnerabilities that are found by ARBITER and responsibly disclosed by the authors during the course of this research project.

- CVE-2018-18311: Integer overflow in Perl before 5.26.3
- Integer overflow in Python-PIL [6]
- Integer overflow in LibTIRPC [7]
- Unchecked setuid in CSound [8]
- Unchecked setuid in ping and ping6 in Inetutils [9]

B Specializing ARBITER

We implemented ARBITER as a general analysis framework that is designed to be easily specializable to detect instances of specific vulnerability classes. In our framework, analysts implement short vulnerability descriptions that specify sources, sinks, and dynamic constraints. ARBITER, in turn, is implemented in roughly 2,000 lines of Python on top of the `angr` binary analysis engine (in which we modified roughly 140 lines of code to implement improvements necessary for ARBITER).

In this section, we describe the effort involved to specialize ARBITER to detect vulnerabilities in the four classes we chose for our evaluation: CWE-131 — "Incorrect Calculation of Buffer Size", CWE-252 — "Unchecked Return Value", CWE-337 — "Predictable Seed in Pseudo-Random Number Generator", and CWE-134 — "Controlled Format String". ARBITER's vulnerability descriptions are created by implementing three functions:

`specify_sources(binary)`: This function should return a set of program locations (e.g., function names or addresses) and variable specifications (for example, the return value, or a specific argument to the function). These locations will be used as sources to find potentially vulnerable flows in ARBITER's static analysis.

`specify_sinks(binary)`: Similar to sources, this specifies a set of variables to be used as sinks to find potential vulnerable flows.

`apply_constraint(state, sources, sink)`: When ARBITER detects, and symbolically analyzes, a potentially vulnerable flow, it uses this function to check the flow against a symbolic vulnerability condition.

These vulnerability specifications are concise enough to include in this section.

CWE-131: Incorrect Calculation of Buffer Size.

cwe131.py

```python
def specify_sources(binary):
    return {} # defaults to function arguments

def specify_sinks(binary):
  return { "malloc": 0 } # first argument to malloc

def apply_constraint(state, sources, sink):
  for source in sources:
    if source.length < sink.length:
      #equalize bit length of source and sink
      source = source.zero_extend(sink.length-source.length)
    state.solver.add(sink < source)
```

CWE-252: Unchecked Return Value.

cwe252.py

```python
def specify_sources(binary):
  # specify return values (index 0 in arbiter) of
  # security-relevant system calls
  return {
    'access': 0, 'chdir': 0, 'chown': 0,
    'chroot': 0, 'mmap': 0, 'prctl': 0,
    'setrlimit': 0, 'stat': 0, 'setuid': 0,
    'setgid': 0, 'setsid': 0, 'setpgid': 0,
    'setreuid': 0, 'setregid': 0, 'setresuid': 0,
    'setresgid': 0
  }

def specify_sinks(binary):
  # arbiter shorthand for the return value
  # of the _caller_ of these functions
  return [
    'access', 'chdir', 'chown',
    'chroot', 'mmap', 'prctl',
    'setrlimit', 'stat', 'setuid',
    'setgid', 'setsid', 'setpgid',
    'setreuid', 'setregid', 'setresuid',
    'setresgid'
  ]

def apply_constraint(state, sources, sink):
  if state.satisfiable(
    extra_constraints=[sources[0] == -1]
  ):
    # target function allows both negative and
    # positive values (indicating absence of
    # checks)
    state.solver.add(sources[0] == 0)
  else:
    # reject the state by making it unsatisfiable
    state.solver.add(False)
```

CWE-134: Controlled Format String.

cwe134.py

```python
def specify_sources(binary):
    return {} # defaults to function arguments

def specify_sinks(binary):
  # the format argument of string formatters
  return {
    'printf': 0, 'fprintf': 1, 'dprintf': 1,
    'sprintf': 1, 'vasprintf': 1, 'snprintf': 2,
    'fprintf_chk': 2, 'dprintf_chk': 2,
    'sprintf_chk': 3, 'vasprintf_chk': 2,
    'asprintf_chk': 2, 'snprintf_chk': 4,
   }

def apply_constraint(state, sources, sink):
  # check for the format string in the ELF
  addr = state.solver.eval(sink, cast_to=int)
  elf_address = \
```

[6] https://github.com/python-pillow/Pillow/pull/3703

[7] http://git.linux-nfs.org/?p=steved/libtirpc.git;a=commit;h=56b780e61ed4bae8b728a600fc5ac8052d0d3582

[8] https://github.com/csound/csound/pull/1335

[9] https://git.savannah.gnu.org/cgit/inetutils.git/commit/?id=02a379763bf651a09b5cb728c1d6b811dc71d021

```
    state.project.loader.find_section_containing(addr)
  if elf_address is not None:
    state.solver.add(False)
```

CWE-337: Predictable Seed in PRNG.

cwe337.py

```
def specify_sources(binary):
  # return value of time()
  return { "time": 0 }

def specify_sinks(binary):
  # first argument of srand()
  return { "srand": 0 }

def apply_constraint(state, sources, sink):
  # no constraints --- purely a data flow problem
  pass
```

C Performance Evaluation

In order to evaluate the performance of ARBITER, we executed the integer overflow detection described in Section 3.1 on all 105 binaries in the well-tested *coreutils* suite. It took a total of 884 seconds for ARBITER to analyze 27,608 functions and locate sources and sinks in 763 of them.

During the data flow recovery step, ARBITER experienced failures in the underlying angr framework for 62 of the 763 functions. For the remaining 701 cases, it successfully extracted data flows to 1,303 allocator call sites. Out of these, 364 had constant sizes (no data dependency), and 939 resulted in paths representing data flows allocator call sites across 567 functions. This phase took a total of 4,585 seconds.

ARBITER then symbolically executed each path with Under-Constrained Symbolic Execution, resulting in 45 failures due to implementation problems, 251 timeouts (with the default 10-minute timeout per flow), 133 failures to find a satisfiable path (i.e., as result of over-approximation of static analysis), and 510 data flow predicates. This step took a total of 15,705 seconds.

Checking these 510 predicates against the VD description resulted in 124 candidate alerts. However, after performing its adaptive false positive filtering to a level of two callers, ARBITER successfully eliminated all 124 false positives.

D Detailed Evaluation Results

We detail the binaries, functions, identified sinks, determined data flows, explored paths, analyzed states, and raised alerts in the large-scale evaluation on CWE-131, CWE-134, CWE-252, and CWE-337 in Table 6.

E Case Studies

For the voracious reader, we present a number of case studies of select bugs identified by ARBITER.

Case Study: Integer Overflow in Perl. ARBITER found a vulnerability that was present in versions of the Perl runtime before 5.28. It could lead to heap overflows and potentially to arbitrary code execution. The relevant source code snippet is shown in Listing 2.

Listing 2 Part of the Perl source code that contains the *Perl_my_setenv* function, where ARBITER detected an integer overflow in line 14. The vulnerability was assigned to CVE-2018-18311.

```
1  #define my_setenv_format(s, nam, nlen, val, vlen) \
2    Copy(nam, s, nlen, char); \
3    *(s+nlen) = '='; \
4    Copy(val, s+(nlen+1), vlen, char); \
5    *(s+(nlen+1+vlen)) = '\0'
6
7  void
8  Perl_my_setenv(pTHX_ const char *nam, const char *val)
9  {
10   ...
11   const int nlen = strlen(nam);
12   const int vlen = strlen(val);
13   char * const new_env = (char*)safesysmalloc(
14     (nlen + vlen + 2) * sizeof(char));
15   my_setenv_format(new_env, nam, nlen, val, vlen);
16   ...
```

Listing 3 Part of the Pillow source code that contains an integer overflow in line 5.

```
1  int
2  ImagingMemorySetBlocksMax(ImagingMemoryArena arena, int blocks_max){
3    ...
4    else if (arena->blocks_pool != NULL) {
5      p = realloc(arena->blocks_pool, sizeof(*arena->blocks_pool) *
6            blocks_max);
7      ...
```

The function *Perl_my_setenv* is called when the user decides to initialize the value of an environment variable. The variables *nam* and *val* are the environment variable's name and value, respectively. In this function, the two 32-bit integers *nlen* and *vlen* are used to store the length of the *nam* and *val* string, respectively. These integers are then added together along with the constant *2*, and the sum is used to allocate a buffer on the heap.

The addition at line 14 in Listing 2 is the point where this arithmetic can lead to an integer overflow, which subsequently leads to a small buffer being allocated. At line 15, the function *my_setenv_format* copies the two strings separately to the newly created buffer, which results in a heap buffer overflow with controlled input. This vulnerability was assigned CVE-2018-18311 and has now been fixed. However, it is interesting to note that this vulnerability was introduced in 2001 and remained undetected until 2018.

Case Study: Integer overflow Vulnerability in Pillow. ARBITER found another vulnerability in the *python-pil* package, which we reported to the maintainers and has since been fixed. The vulnerability occurs in the function *ImagingMemorySetBlocksMax* and a snippet of the relevant code is provided in Listing 3.

The integer overflow vulnerability occurred at line 5 when the 32-bit signed integer *blocks_max* is multiplied with a constant, which is then used as the second argument to the function *realloc*. In a 64-bit environment, the second argument to

CWE	CWE-131	CWE-134	CWE-252	CWE-337
Starting binaries	76,516	76,516	76,516	76,516
Recon binaries / functions	42,611 / 980,758	42,690 / 1,233,827	14,228 / 96,715	3,207 / 4,554
DDA analyzed binaries / functions / flows	41,796 / 788,970 / 1,307,990	42,366 / 1,044,218 / 3,311,716	14,103 / 88,399 / 84,359	3,206 / 4,490 / 5,074
DDA identified binaries / functions / sinks	36,696 / 346,701 / 692,501	26,576 / 192,846 / 429,711	13,485 / 55,388 / 55,388	3,106 / 4,083 / 4,641
UCSE analyzed binaries / functions / paths	29,577 / 209,217 / 145,167	19,410 / 103,926 / 87,383	12,081 / 47,913 / 73,823	2,849 / 3,622 / 3,414
UCSE identified binaries / functions / states	11,495 / 24,781 / 31,436	13,460 / 39,276 / 39,305	1,851 / 2,388 / 4,413	2,239 / 2,661 / 2,741
FP1 binaries / functions / states	1,916 / 2,693 / 3,310	3,013 / 4,891 / 4,894	310 / 686 / 942	247 / 277 / 281
FP2 binaries / functions / states	695 / 928 / 1,037	420 / 489 / 489	71 / 71 / 126	83 / 92 / 92
FP3 binaries / functions / states	270 / 318 / 351	183 / 221 / 222	58 / 58 / 107	42 / 45 / 45
Main binaries / functions / states	121 / 161 / 181	16 / 25 / 39	/ 68 / 77 / 101	328 / 347 / 352
Final alerts	436	158	159	377

Table 6: Summarizing the results of the experiments for 4 templates targeting 4 types of CWEs. This table shows the detailed output for each of ARBITER's analyses. **Main** binaries are those binaries in which a false positive reduction step reached the "main()" function as a caller. In this case, ARBITER reports the alert (as no further FP reduction is possible). Because of how the data was logged, there is an overlap between the binaries/functions/alerts in Main and FP3. The **Final alerts** contains the total number of alerts for each CWE type after filtering out alerts that do not correspond to the list of sinks we use for each CWE type.

Listing 4 Pillow code of the calling context of function *ImagingMemorySetBlocksMax*. The check in line 7 prevents the integer overflow, that ARBITER detected, in a 64-bit environment. On a 32-bit architecture, however, an integer wrap-around can be achieved with a positive value, hence, the vulnerability is triggerable.

```
1  static PyObject*
2  _set_blocks_max(PyObject* self, PyObject* args)
3  {
4    int blocks_max;
5    if (!PyArg_ParseTuple(args, "i:set_blocks_max", &blocks_max))
6      return NULL;
7    if (blocks_max < 0) {
8      PyErr_SetString(PyExc_ValueError,
9        "blocks_max should be greater than 0");
10     return NULL;
11   }
12   ...
13   ImagingMemorySetBlocksMax(&ImagingDefaultArena, blocks_max);
14   ...
```

realloc is a 64-bit value whereas *blocks_max* is a 32-bit value. Since *blocks_max* is a signed integer, it gets sign-extended to a 64-bit value. In this situation, the only way that an integer overflow occurs is if the sign extension results in a large 64-bit value, which in turn only occurs if the initial *blocks_max* is a negative value. However, if we look at the code that invokes this *ImagingMemorySetBlocksMax* function, we see that this vulnerability cannot be triggered. The relevant code is shown in Listing 4.

The *PyArg_ParseTuple* is used to extract a 32-bit integer value from the *args* object and store the result in the *blocks_max* variable. We can see that this value is immediately compared with the value 0, and an error is generated if it is found to be lower. Therefore, there is no possible way of triggering this vulnerability in a 64-bit environment.

However, in a 32-bit environment, the second argument to *realloc* is a 32-bit value, which makes it possible to generate a positive value that can wrap around the 32-bit integer space during the multiplication. We were able to find such a positive value that would result in a value of 0 after multiplication. According to the description of *realloc*, when the second argument is 0, it is equivalent to calling *free*. Without the pointer being set to NULL, this ends up being a Use-After-Free (UAF) scenario. We reported this vulnerability to the maintainers and also submitted a patch, which has since been merged. However, the maintainers argued that since triggering this vulnerability required arbitrary python execution, it could not be considered as a security risk.

Listing 5 Attempting to drop privileges before creating log file inside Csound.

```
1  int set_rt_priority(int argc, const char **argv)
2  {
3    ...
4    ignore_value(setuid(getuid()));
5    ...
6  }
```

Case Study: Unchecked RetVal in Csound. As described in the example in Section 3.2, a usual pattern of dropping privileges in a binary with SETUID capability involves invoking *getuid* to get the actual user ID of the user who started the process. This is immediately followed by invoking *setuid* to change the user ID of the process to the real user ID from the effective user ID. However, when the software does not check to see if *setuid* dropped the privileges successfully, it might lead to privilege escalation.

We found multiple instances of such behaviour in *Csound*. The relevant source code snippet is shown in Listing 5. The the *setuid* is invoked inside the function *set_rt_priority* to drop root privileges. After this function *set_rt_priority* returns, the program then creats a log file that is then written to.

In this case, if the *setuid* fails, the log file that would be left on the machine would be owned by root which is a dangerous situation. Since, disregarding the return values of important API functions like *setuid* can lead to security vulnerabilities, compilers often generate warnings to report a situation where the return value of such an API is unused. However, in the snippet shown in Listing 5, the *ignore_value* macro is used to suppress a compiler generated warning. We have submitted a patch for this vulnerability and it has now been fixed.