ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems

Carter Yagemann
Georgia Institute of Technology

Matthew Pruett
Georgia Tech Research Institute

Simon P. Chung
Georgia Institute of Technology

Kennon Bittick
Georgia Tech Research Institute

Brendan Saltaformaggio
Georgia Institute of Technology

Wenke Lee
Georgia Institute of Technology

Abstract

End-host runtime monitors (e.g., CFI, system call IDS) flag processes in response to symptoms of a possible attack. Unfortunately, the symptom (e.g., invalid control transfer) may occur long after the root cause (e.g., buffer overflow), creating a gap whereby bug reports received by developers contain (at best) a snapshot of the process long after it executed the buggy instructions. To help system administrators provide developers with more concise reports, we propose ARCUS, an automated framework that performs root cause analysis over the execution flagged by the end-host monitor. ARCUS works by testing "what if" questions to detect vulnerable states, systematically localizing bugs to their concise root cause while finding additional enforceable checks at the program binary level to demonstrably block them. Using hardware-supported processor tracing, ARCUS decouples the cost of analysis from host performance.

We have implemented ARCUS and evaluated it on 31 vulnerabilities across 20 programs along with over 9,000 test cases from the RIPE and Juliet suites. ARCUS identifies the root cause of all tested exploits — with 0 false positives or negatives — and even finds 4 new 0-day vulnerabilities in traces averaging 4,000,000 basic blocks. ARCUS handles programs compiled from upwards of 810,000 lines of C/C++ code without needing concrete inputs or re-execution.

1 Introduction

End-host runtime monitors are designed to enforce security properties like control flow integrity (CFI) [1]–[10] or detect anomalous events (system calls [11], segmentation faults [12]–[15]). They can effectively halt attacks that rely on binary exploits and are seeing real-world deployment [16], [17]. However, these systems are designed to react to the *symptoms* of an attack, not the *root cause*. A CFI monitor responds to an invalid control flow transfer, not the buggy code that allowed the code pointer to become corrupted in the first place. A host-based IDS responds to an unusual sequence of system calls, without concern for how the program was able to deviate from the expected behavior model.

Traditionally, symptoms of an attack are easier to detect than root causes. Namely, it is easier to detect that the current state *has* violated a property than to diagnose what lead to that violation. Unfortunately, this has led security professionals to adopt brittle stopgaps (e.g., input filters [18]–[21] or selective function hardening [22]), which can be incomplete or incur side effects (e.g., heavyweight instrumentation [23]). Ideally, the developers that maintain the vulnerable program must fix the code and release a patch, but this creates a conundrum: *where is the bug that led to the detected attack?*

Unfortunately, the journey from a detected attack to a patch is rarely easy. Typical attack artifacts, like crash dumps [24] or logs [25]–[35], contain partial, corruptible data [36]–[42] with only the *detection* point marked. Concrete inputs may reproduce the symptoms in the production environment, but raise privacy concerns [24] and rarely work for developers [43], [44]. Worse still, developers are known to undervalue a bug's severity [45] or prioritize other (better understood) issues [46].

Seeking a better solution, we propose a root cause analysis that considers "what if" questions to test the impact of particular inputs on the satisfiability of vulnerable states. The tests are vulnerability-class-specific (e.g., buffer overflows) and enable the analysis to localize vulnerabilities and recommend new enforceable constraints to prevent them, essentially suggesting a patch to developers. Analysis is conducted over the control flow trace of the program flagged by the end-host monitors, testing at each state "what if" any of the vulnerability tests could be satisfied. Notice that this is a divergence from the traditional mindset of replaying [47]–[49] or tainting [21], [50]. For example, instead of tainting a string that caused a stack overflow, the developers would most directly benefit from knowing which code block caused the corruption and what additional constraints need to be enforced upon it.[1]

Armed with vulnerability-class-specific satisfiability tests, we turn our attention to efficiently collecting control flow

[1] Such analysis could also merge redundant alerts stemming from the same bug producing varying symptoms, improving alert fatigue [51]–[53].

traces in production end-hosts, which is challenging due to strict performance expectations. Interestingly, we find that readily available, hardware-supported, processor tracing (PT)[2] offers a novel avenue towards efficient recording. Specifically, we leverage the capability of Intel® PT to design a kernel module that can *efficiently capture* the control flow of user programs, storing and forwarding it to an analysis system if the end-host runtime monitor flags the process. Notably, this avoids recording concrete data or attempting to re-execute the program.

We have implemented a system called ARCUS[3] — an automated framework for localizing the root cause of vulnerabilities in executions flagged by end-host runtime monitors. We have evaluated our ARCUS prototype using 27 exploits targeting real-world vulnerabilities, covering stack and heap overflows, integer overflows, allocation bugs like use after free (UAF) and double free (DF), and format string bugs, across 20 different commodity programs. Surprisingly, ARCUS also discovered 4 new 0-day vulnerabilities that have been issued 3 CVE IDs, demonstrating an ability to find neighboring programming flaws.[4] ARCUS demonstrates impressive scalability, handling traces averaging 4,000,000 basic blocks from complicated programs and important web services (GIMP, Redis, Nginx, FTP, PHP), compiled from upwards of 810,000 source lines of C/C++ code. It also achieves 0 false positives and negatives in analyzing traces taken of the over 9,000 test cases provided by the Juliet and RIPE benchmarks for our implemented classes. We show that tracing incurs 7.21% performance overhead on the SPEC CPU 2006 benchmark with a reasonable storage requirement. To promote future work, we have open source ARCUS and our evaluation data.[5]

2 Overview

ARCUS' analysis begins when an end-host runtime monitor flags a running process for executing some disallowed operation. Three classes of such systems are widely deployed today: CFI monitoring [1]–[10], system call/event anomaly detection [11], and segmentation fault/crash reporting [12]–[15]). However, ARCUS is not dependant on *how or why the process was flagged*, only that it was flagged. Notice that ARCUS must handle the fact that these systems detect attacks at their *symptom* and not their onset or root cause. In our evaluation, we tested alongside a CFI monitor [1] and segmentation fault handler, both of which provide delayed detection. ARCUS can easily be extended to accept triggers from any end-host runtime monitor.

```
1  int openhost(const char *hname, ...) {
2    char *cp;
3    char name[256];
4
5    cp = hname;
6    if (*cp == '[') {
7      cp++;
8      for (i = 0; *cp && *cp != ']'; cp++, i++)
9        name[i] = *cp; //buffer overflow
10     if (*cp == ']') {
11       name[i] = '\0';
12       hname = name;
13     } else return 0;
14   /* [...] */
```

Figure 1: CVE-2018-12327 in `ntpq`. A stack overflow occurs if there is no '`]`' within the first 257 characters of `hname`.

2.1 Real-World Example

We will briefly walk through how to apply our proposed solution to a real vulnerability: CVE-2018-12327. We pick this example because the bug is concise and straightforward to exploit. Conversely, a case study containing thousands of intermediate function calls is presented in Section 4.5. We will stay at a high level for this subsection and revisit the same example in greater detail in Subsection 3.2.

CVE-2018-12327 is a stack overflow bug exploitable in `ntpq` to achieve arbitrary code execution. The vulnerability exists because there is no check for the length of the relevant command line argument. We will follow the *source code* in Figure 1 for simplicity, but the actual analysis is on *binaries*.

Assume the attacker can manipulate the arguments passed to `ntpq`, allowing him to overwrite the stack with a chain of return addresses that will start a reverse shell — a typical example of *return-oriented programming* (ROP). When `ntpq` starts, the ARCUS kernel module snapshots the program's initial state and configures PT. The malicious input triggers the bug, and a shell is created. A runtime monitor determines that the shell spawning is anomalous and flags the program, causing the kernel module to send the snapshot and trace for analysis.

The analysis sequentially reconstructs a symbolic program state for each executed basic block. All input data, including command line arguments, are symbolized. As the states are stepped through, a plugin for each implemented bug class checks for memory violations (Subsection 3.3). Since the attacker's input is symbolic, when the buggy code corrupts the stack, the return pointer will also become symbolic. The return causes the program counter to become symbolic, which is detected by the stack overflow module as a vulnerability.

ARCUS now switches to localizing the root cause. It identifies the symbolic instruction pointer in memory and finds the prior state that made it become symbolic (compiled from line 9). By examining the control dependencies of this state, ARCUS automatically identifies the guardian basic block that

[2] Available in Intel®, AMD®, and ARM® processors.

[3] Analyzing Root Cause Using Symbex.

[4] We reported new vulnerabilities to MITRE for responsible disclosure.

[5] https://github.com/carter-yagemann/ARCUS

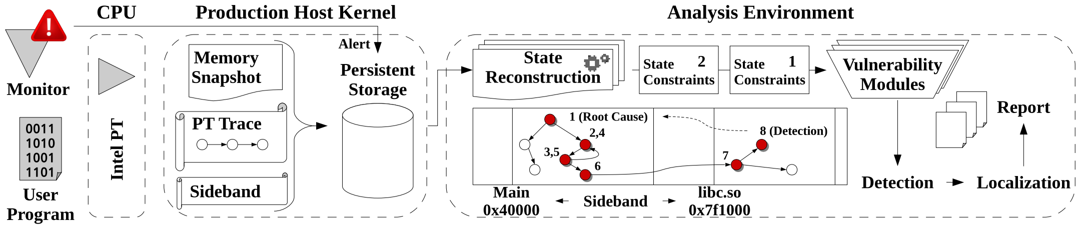


Figure 2: ARCUS architecture. The user program executes in the end-host while the ARCUS kernel module snapshots and traces it using Intel PT. When a runtime monitor flags a violation or anomaly, the data is sent to the analysis environment where symbolic states are reconstructed, over which the modules detect, localize, and report vulnerabilities.

decides when the relevant loop will exit (compiled from line 8). ARCUS determines the loop could have exited sooner and checks what would happen if it did (the "what if" question, elaborated on in Subsection 3.2). ARCUS verifies that this alternative state does not have a symbolic return pointer, compares the resulting data constraints to those in the compromised program state, and spots the contradiction — a special delimiter character at a particular offset of an input string. It uses this to automatically recommend a new constraint to enforce at the guardian to fix the overflow.

As output, the human analyst automatically receives a report containing: 1) the basic block that corrupted memory, 2) the guardian that failed to protect against the exploit, and 3) a recommended fix for the guardian.

2.2 Threat Model

We consider attacks against user programs and assume that the kernel and hardware in the production system are trustworthy, which is reasonable given that Intel PT is a hardware feature that writes directly to physical memory, bypassing all CPU caches, configurable only in the privileged CPU mode. This is consistent with prior security work relying on Intel PT [1], [2], [54], [55]. We do not alter user space programs in any way. The kernel module also provides a secure way to store and forward recorded data to an analysis system, which may be a separate server for extra isolation.

We expect attackers to target the production system's programs, but not have direct access to the analysis. We focus on program binaries without assuming access to source code or debug symbols.[6] Consequently, we cannot handle all *data-only* attacks (e.g., selectively corrupting a flag), which may require accurate type information. However, ARCUS can be extended in future work to incorporate this.

3 Design

ARCUS consists of two general components, shown in Figure 2. A kernel module snapshots the initial state of the monitored program and collects its subsequent control flow via PT (Subsection 3.4). The data is recorded to secure storage reserved by the kernel module and if an alarm is raised by a runtime monitor, it is transmitted to the analysis system, which may reside in a separate server. ARCUS is compatible with any end-host runtime monitor that can flag a process ID. We use an asynchronous CFI monitor [1] and a segmentation fault handler in our evaluation for demonstration.

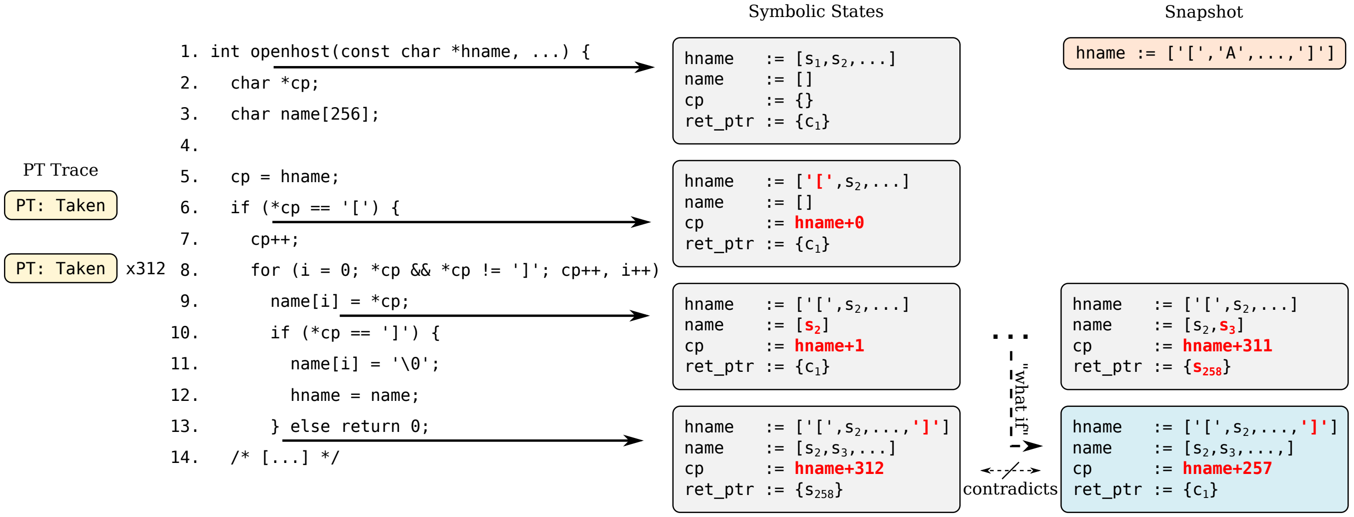
The analysis is facilitated using symbolic execution with pluggable modules for different classes of bugs (Subsection 3.3). This serves to reconstruct the possible data flows *for a single path*, which enables the system to spot vulnerable conditions (e.g., a large input integer causing a register to overflow) and consider "what if" questions to automatically find contradictory constraints that prune the vulnerable state (Subsection 3.2). ARCUS then automatically recommends places in the binary to enforce these constraints so that developers can quickly understand and patch the root cause.

3.1 Symbolic Execution Along Traced Paths

Once an alarm is raised by a monitor, ARCUS will construct symbolic program states from the data sent by the kernel module. Our insight is to use symbolic analysis, but with special consideration to avoid its greatest shortcoming: state explosion. Put briefly, symbolic analysis treats data as a combination of *concrete* (one possible value) and *symbolic* (multiple possible values) data. As the analysis explores different paths in the program, it places *constraints* on the symbolic data, altering their set of values. In this way, symbolic analysis tracks the possible data values that can reach a program state.

We use symbolic analysis *not* to statically explore all possible paths, as is the typical use case, but to instead consider *all possible data flows over one particular path*. To do this, we symbolize all input data that could be controlled by the attacker (command line arguments, environment variables, files, sockets, and other standard I/O) and only build constraints for the path that was traced. This sidesteps the biggest problem with performing analysis in a vacuum — state explosion — by leveraging the execution trace leading up to the end-host runtime monitor's alert.

[6] However, we reference source code in our explanations and figures whenever possible for brevity and clarity.



```
 1. int openhost(const char *hname, ...) {
 2.   char *cp;
 3.   char name[256];
 4.
 5.   cp = hname;
 6.   if (*cp == '[') {
 7.     cp++;
 8.     for (i = 0; *cp && *cp != ']'; cp++, i++)
 9.       name[i] = *cp;
10.       if (*cp == ']') {
11.         name[i] = '\0';
12.         hname = name;
13.       } else return 0;
14.   /* [...] */
```

Figure 3: Revisiting CVE-2018-12327 in more detail. Part of the snapshot and constraints tracked by ARCUS are shown on the right with registers and addresses substituted with variable names for clarity. PT is on the left.

3.2 "What If" Questions

Reasoning over symbolic data also enables ARCUS to consider "what if" questions, which is a key novelty in our root cause analysis. We now revisit CVE-2018-12327 (introduced in Subsection 2.1) to show how ARCUS uses "what if" questions in detail. In Figure 3, part of the snapshot (orange box) and constraints tracked by ARCUS (grey boxes) are shown on the right. We substitute registers and memory addresses with variable names for clarity, but keep in mind that ARCUS operates on binaries without needing debug symbols or source code. A part of the PT trace (yellow boxes) is shown on the left with the source code in the center. We use square brackets to denote array contents and curly to list the possible values for a variable. The notation s_i is for unconstrained symbolic data and c_i is for concrete constants. ret_ptr is the return pointer.

ARCUS starts by replacing the attacker-controlled data in the snapshot with symbolic variables. hname points to a command line argument, which is why its contents become symbolic. As ARCUS symbolically executes the program, it follows the PT trace, which says to take the branch at line 6 and to repeat the loop 312 times. As the loop iterates, cp increments, and name is filled with symbolic values copied from hname. By the time line 14 is reached, the return pointer has been overwritten with an unconstrained symbolic value. When the function returns, the program counter becomes symbolic, which means the attacker is capable of directly controlling the program's execution via crafted command line arguments. This is a vulnerability that triggers the stack overflow module in ARCUS to begin root cause analysis.

The full algorithm for this vulnerability class is presented in Subsection 3.3, so for brevity we will focus on the "what if" question, which comes into play after ARCUS has located the symbolic state prior to ret_ptr being corrupted. ARCUS revisits this state and discovers there is another possible path where the loop exits sooner, which requires $\texttt{cp} \leq \texttt{hname}+257$ and the 257th character in hname to be ']'.

What if this path were to be taken by the program? The resulting constraints would contradict the ones that led to the corrupted state, which requires ']' to occur in hname no sooner than offset 258. Thus, by solving the "what if" question, ARCUS has automatically uncovered a fix for the vulnerability. Subsection 3.3 covers how the module then determines where to enforce the new data constraints to make the recommendation more concise and practical. Note that even after applying the recommended fix, line 14 of the program is still reachable. However, because the newly enforced constraints contradict the compromised state, the code can no longer be executed in the context that would give rise to the observed overflow.

3.3 Analysis Modules

In this subsection, we expand on our methodology from Subsections 3.1 and 3.2 to describe how serious and prevalent classes of vulnerabilities can be analyzed using ARCUS. Each class has a refined analysis strategy and definition of root cause based on our domain expertise. In our prototype, each technique is implemented as a pluggable module, summarized in Table 1. Each module description concludes with a list of contents generated by ARCUS in its reports.

Stack & Heap Overflow. The stack and heap overflow module focuses on analyzing control flow hijacking (recall that data-only attacks are out of scope, Subsection 2.2), which requires the adversary to gain control over the program

Table 1: ARCUS Modules Summary

Module	Locating Strategy	Root Cause
Stack Overflow	Symbolic PC	Control Dep.
Heap Overflow	Symbolic PC	Control Dep.
Integer Overflow	Overflowed Reg/Mem	Overflow Site
UAF	R/W Freed Address	Control Dep.
Double Free	Track Frees	Control Dep.
Format String	Symbolic Arguments	Data Dep.

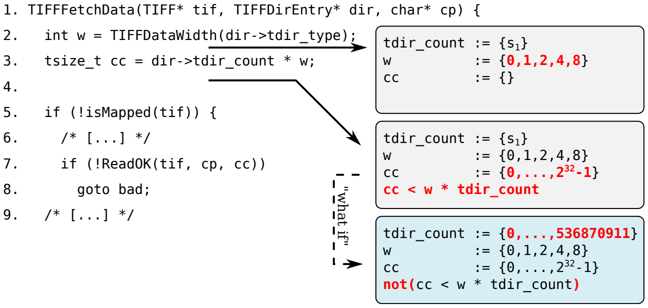
counter. As ARCUS reconstructs all the intermediate states along the executed path, the module checks whether the program counter has become symbolic. If it has, this means data from outside the program can exert direct control over which code the program executes, which is indicative of control hijacking.

From this point, the module looks at the previous state to determine what caused symbolic data to enter the program counter. Since hijacking can only occur at indirect control flow transfers, this previous state must have executed a basic block ending in a return, indirect call, or indirect jump. The steps we define for root cause analysis are: 1) identify the code pointer that became symbolic, 2) identify the basic block that wrote it, 3) find basic blocks that control the execution of the write block, and 4) test whether additional constraints at these blocks could have diverted the program away from the buggy behavior (i.e., by introducing a constraint that would contradict the buggy state).

To accomplish the first task, the module uses backward tainting over the previously executed basic block, lifted into an intermediate representation (IR), to identify the registers and then the memory address used to calculate the code pointer. The implementation details are in the Subsection 3.7. Once identified, the module iterates backwards through the previously reconstructed states to find the one where the data contained at the identified address changes, which reveals the state that corrupted the pointer. We coin this the *blame state*.

The next step is to identify the basic blocks that control it, which we refer to as *guardians*. The module uses forward analysis over the reconstructed states to generate a control dependency graph (CDG) and find them.[7] If there are guardians for the blame state, the closest one is picked in terms of shortest path, and the prior state to execute this code is revisited to see if there exists another branch whose constraints contradict the blame state (solving the "what if" question from Subsection 3.2). If contradicting constraints are found, ARCUS recommends enforcing them at the guardian. Otherwise, only the blame state is reported because an entirely new guardian is required.

For heap overflows, ARCUS needs to ensure that the heap objects are allocated exactly as they were in the flagged execution, which requires careful designing. We elaborate on the details in Subsection 3.5.



```
1. TIFFFetchData(TIFF* tif, TIFFDirEntry* dir, char* cp) {
2.   int w = TIFFDataWidth(dir->tdir_type);
3.   tsize_t cc = dir->tdir_count * w;
4.
5.   if (!isMapped(tif)) {
6.     /* [...] */
7.     if (!ReadOK(tif, cp, cc))
8.       goto bad;
9.   /* [...] */
```

Figure 4: CVE-2006-2025. Attacker controls the TIFF image and thus tdir_count, which can be used to overflow cc. ARCUS automatically finds a new constraint to prevent it.

Report: Blame state and, if found, the guardian to modify and new constraints to enforce.

Integer Overflow & Underflow. The two key challenges with detecting integer overflows and underflows (referred to collectively as overflows for brevity) are: 1) inferring the signedness of register and memory values in the absence of type info and 2) avoiding false positives due to intentional overflowing by developers and compilers.

To conservatively infer signedness, the module uses hints provided by instruction semantics (e.g., zero vs. signed extending [56]), and type info for arguments to known standard library functions ("type-sinking" [57]). If the signedness is still ambiguous for an operand, the arithmetic operation is skipped to err on the side of false negatives.

If an operation can overflow, according to the accumulated data constraints, the result register is flagged and subsequent stores and loads are tracked by the module. However, this is not immediately reported as a bug because the overflow may be intentional (second challenge). Instead, a bug is only reported if flagged data is passed to another function (i.e., following a call or ret instruction). The intuition is that when data crosses a function boundary, it is likely that the receiver did not consider the possibility of receiving overflowed integers, leading to violated assumptions and bugs. Prior work has measured this phenomenon [58].

Figure 4 illustrates how the module handles CVE-2006-2025, showing source code for clarity. In this case, an adversary can craft a TIFF image to overflow the register holding cc (defined at line 3) and pass it to ReadOK at line 7. Since cc is the product of two unsigned values, cc < w ∗ tdir_count should not be possible, yet at line 4 the module discovers it is satisfiable, indicating cc can overflow. When cc is then passed to ReadOK, the module flags the bug.

To recommend a fix, the module solves the "what if" question: *what if the prior constraint was not satisfiable?* This requires an additional data constraint to be placed on tdir_count. The module includes this in its report along with the basic block that overflowed cc and the basic block

[7] These graph algorithms are readily available in projects like angr.

that passed `cc` to `ReadOK`.

Report: Basic block and IR statement that overflowed the variable, recommended constraints, and basic block that passed the overflowed variable to another function.

Use After Free & Double Free. The UAF and DF modules monitor all calls to allocation and free functions, which we assume to know the semantics of in advance. When an allocation call is reached, the size argument is extracted and the returned pointer is evaluated to a concrete value to maintain a list of currently allocated buffers. When a free is reached, the corresponding entry is moved from the allocation list to a freed buffers list. Subsequent allocations can move freed entries back to the allocation list, maintaining mutually exclusive sets. For each state, addresses accessed by memory operations are checked against the freed list to detect the occurrence of UAF, upon which the module reports the starting address, size, and accessed offset. Similarly, the DF module detects freeing of entries already in the freed list. A CDG from the free site to the violating block determines and reports negligent guardians.

Report: Address, size, and offset (if applicable) of the violated buffer. The freeing and violating basic blocks, along with a partial CDG for the path between them.

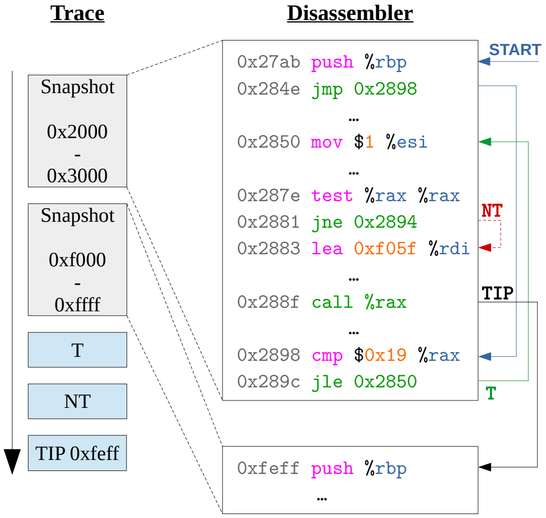
Format String. Programming best-practice is to always create format strings as constant values in read-only memory. Unfortunately, buggy programs still exist that allow an attacker to control a format string and achieve arbitrary reads or writes. As the analysis reconstructs program states, this module checks for states entering known format string functions (e.g., `printf`) and verifies that: 1) the pointer to the format string is concrete, as it should be if it resides in read-only memory, 2) the string's contents are completely concrete, and 3) all the additional arguments point to mapped memory addresses. If any of these criteria are violated, the module knows data from outside the program can directly influence the format string function, which is a vulnerability.

Once located, the module locates the violating symbolic data in memory and examines prior states to find the one that wrote it. This is the blame state for this category of vulnerability. Since format strings should not be writable in the first place, no further analysis is necessary.

Report: Contents of the symbolic string, the basic block that wrote it, and where it was passed to a format function.

3.4 Capturing the Executed Path

Analyzing the execution flagged by an end-host runtime monitor, which may reside in a different system, requires an efficient way of tracing the program without relying on instrumentation or binary modifications that could degrade performance or be targeted by the attacker. Our solution is to employ a kernel module to manage PT. For simplicity, we will focus on Intel PT, but other modern processors come with their own hardware implementations.



```
0x27ab push %rbp
0x284e jmp 0x2898
        …
0x2850 mov $1 %esi
        …
0x287e test %rax %rax
0x2881 jne 0x2894
0x2883 lea 0xf05f %rdi
        …
0x288f call %rax
        …
0x2898 cmp $0x19 %rax
0x289c jle 0x2850

0xfeff push %rbp
        …
```

Figure 5: Using the trace (left), with snapshot and PT packets, to recover the executed sequence of instructions (right).

A trace captures the sequence of instructions executed by the CPU, which is large given that modern processors execute millions of instructions per second. To be efficient, Intel PT assumes that the auditor knows the memory layout of the audited program, which our kernel module prepends to the trace as a snapshot, shown on the left side of Figure 5 as grey packets. The kernel module also captures and inserts dynamically generated code pages between PT data, allowing complex behaviors to be followed (e.g., JIT). With this, all the auditor needs from the PT hardware is which path to follow when a branch is encountered, shown on the left in blue. For conditional branches, a single *taken-not-taken* bit is recorded. For indirect control flow transfers (return, indirect call, and indirect jump) and asynchronous events (e.g., interrupts, exceptions), the destination is recorded.

Intel PT is configured using *model specific registers* (MSRs) that can only be written and read while the CPU is in privileged mode. Since only the kernel executes in this mode, only it can configure Intel PT. The trace is written directly into memory at *physical* addresses specified during configuration, meaning the kernel can make this data inaccessible to all other processes. Intel PT bypasses all caches and memory translation, which minimizes its impact on the traced program. When the buffer allocated for tracing is filled, the CPU raises a *non-maskable interrupt* (NMI), which the kernel module handles immediately so no data is lost.

Challenges with PT & Symbolic Execution. Intel PT tries to be as efficient as possible in recording the executed control flow. As a result, only instructions that produce branching paths yield trace packets, which excludes instructions for *re-*

Table 2: Symbolically Executing CISC Repeat Instructions

Type	Common Usage	Strategy
`rep movs`	String Copy	Maximize Iterations
`rep stos`	Memory Initialization	Maximize Iterations
`rep cmps`	String Search (presence)	Symbolize Register
`rep scas`	String Search (offset)	Symbolize Register

peat string operations — used to speed up common tasks. For example, `rep mov` sequentially copies bytes from one memory location to another until a condition is met and `repnz scas` can be used as a replacement for `strlen`. These instructions encode an entire traditional loop into a single statement.

When memory is concrete, these complex instructions are deterministic, so Intel PT does not record how many times they "repeat." This creates a problem for symbolic execution because if these instructions encounter symbolic data in memory or registers, the state will split and the trace will not have information on which successor to follow.

Our solution is to take the path that will most likely lead to a vulnerability, which depends on the type of repeat instruction, shown in Table 2. Three repeat types are excluded (`ins`, `outs` and `lods`) because they are typically used by kernel drivers and not user space programs. For move (`movs`) and store (`stos`), the analysis follows the maximum possible iterations given the symbolic constraints to check for overflow bugs. For comparison (`cmps`) and scanning (`scas`), the analysis skips to the next instruction (i.e., it executes zero iterations) and symbolizes the results register. The constraints for this register depend on the instruction. For example, `repnz scasb` in 64-bit mode scans memory, decreasing `RCX` by 1 for each scanned byte, until either `RCX` becomes 0 or the value stored in `AL` is encountered. The analysis therefore constrains `RCX` to be between 0 and its starting value.

3.5 Snapshots & Memory Consistency

Symbolic execution requires an initial memory state to start its analysis from, which can be created with a custom loader or from a snapshot. The distinction is usually minor, but ends up being vital for ARCUS because it has to follow the path recorded by PT, as opposed to generally exploring the program. We discover that snapshots *are essential* to ARCUS because native loaders have complicated undocumented behaviors that the custom loaders are likely to contradict, creating inconsistencies in memory.

One such discrepancy is in how they resolve *weak symbols*, which can be resolved to one of several possible locations depending on the execution environment. For example, `libc` contains a weak symbol for `memcpy`, which is resolved to point at the most efficient implementation for the processor model. By our count, out of the 2,211 function symbols in `glibc` version 2.28, 30% are weak symbols. Additionally, shared objects can choose to implement their own resolver functions, invoked by the loader, to decide values.[8]

Our solution is for the kernel module to save a concrete snapshot of the program's user space at its entry point — after the initial dynamic loading is complete — and whenever a new thread or process is created. This captures the environment variables, command line arguments, and current *program break* pointer, the latter of which is important for heap placement.

Allocation Consistency. Analyzing attacks requires special care with replicating the spacing and absolute position of dynamically allocated buffers. Inconsistencies could cause overflows between objects or exploited writes to not be reproducible in the analysis.

The solution is to capture the *program break* (`brk`) pointer in the snapshot, which marks the end of the program's data segment. When functions like `malloc` do not have enough space to allocate a new buffer, they make a system call to move the break. Consequently, all dynamically allocated objects are placed relative to the starting position of the break. Therefore, by starting with the same break and following the trace, ARCUS can ensure a consistent layout.

3.6 Performance Constraints

We prioritize performance in our design, but acknowledge that storage is also a concern for long running programs, to which we create two policies. For task-oriented workers, snapshots are taken as the kernel creates them and the oldest snapshots are discarded if a user defined threshold is exceeded. If a long living thread exceeds the threshold, a snapshot is retaken and the oldest data is discarded. This introduces potential false negatives due to truncation, but we demonstrate useful results with practical thresholds in Section 4 and leave improvements to future work.

Since the analysis is performed offline only after an alarm is raised, we relax the performance requirements of the analysis system. Our evaluation shows real vulnerabilities are analyzed in minutes, which is sufficient for practical use.

3.7 Vex IR Tainting

Algorithm 1 shows how we perform backwards tainting on VEX IR lifted from binary code to identify the registers and memory addresses used to calculate a chosen temporary variable. We start by tainting the chosen variable and iterate backwards over the prior statements. Any registers used to store tainted variables (`Put`) become tainted. Whenever tainted variables are assigned a value (`WrTmp`), any registers, memory addresses, or additional variables used to produce the value (i.e., operands) also become tainted. `EvalTmp` uses the symbolic execution engine to resolve memory address pointers.

[8] Example: https://sourceware.org/glibc/wiki/GNU_IFUNC.

```
Input: VEX IR statements $S$ starting from last executed.
Tmp $n$ to taint initially.
Result: Addresses $A$ and registers $R$ used to calculate $n$.
$A \leftarrow \emptyset$
$R \leftarrow \emptyset$
$T \leftarrow \{n\}$
foreach $s$ in $S$ do
    if Type($s$) is Put and Type($s.data$) is RdTmp then
        if $s.data.tmp \in T$ then
            $R \leftarrow R \cup \{s.register\}$
        end
    end
    if Type($s$) is WrTmp and $s.tmp \in T$ then
        foreach $a$ in $s.data.args$ do
            if Type($a$) is Get then
                $R \leftarrow R \cup \{a.register\}$
            end
            if Type($a$) is RdTmp then
                $T \leftarrow T \cup \{a.tmp\}$
            end
            if Type($a$) is Load then
                $A \leftarrow A \cup$ EvalTmp($a.address$)
            end
        end
    end
end
```

Algorithm 1: Tainting algorithm to obtain the registers and addresses used to calculate a VEX IR temporary variable.

To taint multiple basic blocks, we clear T between blocks while persisting A and R.

4 Evaluation

We aim to answer the following questions in our evaluation:

1. *Is ARCUS accurate at detecting bugs within our covered classes?* We perform several micro-benchmarks with a ground truth set of over 9,000 test cases from the RIPE [59] and Juliet [60] suites. This ground truth allows us to verify that ARCUS can find root causes for vulnerabilities with 0 false positives and negatives (Subsection 4.1).
2. *Can ARCUS locate and analyse real-world exploits?* We craft, trace, and have ARCUS analyze exploits for known CVEs and EDBs in real programs. ARCUS successfully handles 27 exploits and even discovers 4 new 0-day vulnerabilities, which we examine in additional case studies (Subsections 4.2 and 4.5).
3. *Are ARCUS' root cause reports consistent with real-world advisories and patches?* We manually verify that ARCUS' root cause reports are consistent with public disclosures and, where available, official patches (Subsection 4.3).
4. *Is ARCUS feasible to deploy in terms of runtime and storage overhead?* We measure the performance and storage overheads of tracing programs using the SPEC CPU 2006 benchmark and Nginx (Subsection 4.4).

Table 3: RIPE and Juliet Test Cases

Overall Results (Detection by ≥ 1 Strategies)

RIPE	TP	TN	FP	FN	Acc.
BSS	170	170	0	0	100%
Data	190	190	0	0	100%
Heap	190	190	0	0	100%
Stack	260	260	0	0	100%

Juliet	TP	TN	FP	FN	Acc.
CWE-134	1,200	2,600	0	0	100%
CWE-415	818	2,212	0	0	100%
CWE-416	393	1,222	0	0	100%

By Locating Strategy (RIPE)

Symbolic IP	TP	TN	FP	FN	Acc.
BSS	154	170	0	16	95.3%
Data	171	190	0	19	95.0%
Heap	154	190	0	36	90.5%
Stack	211	260	0	49	90.6%

Int Overflow	TP	TN	FP	FN	Acc.
BSS	60	170	0	110	67.6%
Data	60	190	0	130	65.8%
Heap	60	190	0	130	65.8%
Stack	150	260	0	110	78.8%

By Locating Strategy (Juliet)

Symbolic Args.	TP	TN	FP	FN	Acc.
CWE-134	1,200	2,600	0	0	100%

Track Frees	TP	TN	FP	FN	Acc.
CWE-415	818	2,212	0	0	100%

R/W Freed Addrs.	TP	TN	FP	FN	Acc.
CWE-416	393	1,222	0	0	100%

Experimental Setup & Runtime Monitor Selection. We use 2 distinct servers to represent the production and analysis systems, each running Debian Buster and containing an Intel® Core™ i7-7740X processor, 32GB of memory, and solid state storage. To serve as end-host runtime monitors, we use an open source CFI system [1] and our own segmentation fault handler. The former is used for the exploits that leverage code reuse attacks and the latter for crashes. We pick this particular CFI monitor because it is asynchronous and only guarantees detection of control flow violations by the next system call, which requires ARCUS to handle traces containing activity past the initial exploit.

4.1 Accuracy on Micro-Benchmarks

Before deploying ARCUS on real-world programs, we evaluate on benchmark test cases where there is known ground truth for the location and behavior of every bug. This is necessary in order to measure false negatives (i.e., executions where a bug is triggered but ARCUS yields no report) and cannot be known for real-world programs.[9] False positives are measurable by manually reviewing reports.

Dataset & Selection Criteria. For the overflow modules (stack, heap, and integer), we use the complete RIPE [59] benchmark, which systematically exploits the provided test binary with different bugs (`memcpy`, `strlen`, etc.), strategies (ROP, code injection, etc.), and memory locations (stack, heap, etc.). We port the benchmark to 64-bit and manually create a second patched (bug-free) version of the test binary to measure false positives (FPs), false negatives (FNs), true positives (TPs) and true negatives (TNs). RIPE yields 810 working exploits in our environment.

RIPE does not contain tests for UAF, double free, or format string bugs. We address this shortcoming with the NIST C\C++ Juliet 1.3 suite [60], which contains 2,411 buggy and 6,034 bug-free binaries for CWE-416 (UAF), CWE-415 (double free), and CWE-134 (format string). These are all the test cases provided by Juliet for these CWEs.

Results. As presented at the top of Table 3, ARCUS correctly analyzes all the test cases across all suites with no FPs or FNs. That is, each TP is detected by at least 1 module and TN by none. We manually verify that the root cause reports for the TP cases correctly identify the buggy functions and the recommendations prevent the memory corruptions.

On closer investigation, we realize that ARCUS is so accurate on the RIPE cases because there are multiple opportunities for detecting overflows. For example, an integer overflow that corrupts a return pointer can be detected either by the integer overflow module when the register wraps around or by the stack overflow module when the pointer is overwritten. Detecting either behavior (or both) yields an accurate report. Based on this observation, we present the middle and bottom portions of Table 3, which separates the RIPE and Juliet results by the locating strategies from Table 1. For the modules tested by the Juliet cases, their capabilities do not overlap and yield the same numbers as in the overall table. For the strategies relevant to RIPE, we discover that the symbolic IP detection is 92.9% accurate, on average, whereas the integer overflow detection is 69.5%. The latter is expected given the challenges described in Subsection 3.3, like inferring signedness in binaries. We observe that the accuracy is consistent across exploit locations for symbolic IP (4.8% variation), but less so for integer overflow (13%) where it performs better on stack-based tests. Since each strategy yields 0 FPs, their capabilities compliment each other, covering their individual weaknesses and enabling ARCUS to operate effectively.

4.2 Locating Real-World Exploits

With ARCUS verified to be working accurately on the microbenchmarks, we turn our attention to real-world exploits.

Dataset & Selection Criteria. We select our vulnerabilities starting with a corpus of *proof of compromises* (PoCs) gathered from the LinuxFlaw [78] repository and Exploit-DB [79], distilled using the following selection procedure:

1. First, we filter PoCs pertaining to bug classes not covered by our modules (Subsection 3.3).
2. Next, we filter PoCs that fail to trigger in our evaluation environment.
3. Finally, for PoCs targeting libraries (e.g., `libpng`), we select a large real-world program that utilizes the vulnerable functionality (e.g., GIMP) for evaluation.

In total, we consider 34 PoCs pertaining to our covered bug classes (Step 1). Of these, 7 failed to trigger and were filtered (Step 2). The primary cause of failure is older PoCs written for 32-bit that cannot be converted to 64-bit. We decide to use GIMP for evaluating image library CVEs, GOOSE Publisher for CVE-2018-18957, exif for CVE-2007-2645, and PHP for CVE-2017-12858 (Step 3).[10]

This yields PoCs targeting 27 unique vulnerabilities across 20 programs, covering a diverse range of multimedia libraries, client applications, parsers, and web services. Some are commonly evaluated in related work (e.g., libexif [80]), whereas others align with our motivation of protecting production servers (e.g., nginx, ftp) and require ARCUS to handle more complex behaviors like multi-threading, inter-process communication, and GUIs (e.g., GIMP). For vulnerabilities that lead to arbitrary code execution, we develop the PoCs into exploits that use code reuse attacks like ROP. We create crashing exploits only as a last resort.

Results. Table 4 shows that our system is able to successfully localize all 27 exploited vulnerabilities. Surprisingly, ARCUS also uncovers 4 new 0-day vulnerabilities — 3 issued CVE IDs — that are possible to invoke along the same control flow path, bringing the total count to 31. An example of how this occurs is presented in Subsection 4.5. For exploited libraries evaluated in the context of a larger program (e.g., CVE-2004-0597), we show the traced program's name alongside the library.

Table 4 includes the number of basic blocks recorded in each trace ("# BBs" column) and size in megabytes ("Size (MB)" column). Traces range from 53,000 basic blocks to over 78,000,000. Sizes are from 600 KB to 56 MB. The larger sizes correlate with programs containing GUIs and complex plug-in frameworks.

[9] If we knew the location and behavior of every bug in real-world programs, we could produce new versions that are guaranteed to be bug-free, which is obviously not possible with existing techniques.

[10] We could not find larger programs in the Debian repositories that trigger CVE-2007-2645 or CVE-2018-18957.

Table 4: System Evaluation for Real-World Vulnerabilities

CVE / EDB	Type	Program	# BBs	Size (MB)	ΔRoot Cause	ΔAlert	Located	Has Patch	Match
CVE-2004-0597	Heap	GIMP (libpng)	41,625,163	56.0	247	1	Yes	[61]	Yes[†]
CVE-2004-1279	Heap	jpegtoavi	67,772	0.65	26,216	1	Yes	No	-
CVE-2004-1288	Heap	o3read	74,723	0.65	33,211	1	Yes	[62]	Yes
CVE-2009-2629	Heap	nginx	300,071	1.10	28	33,824	Yes	[63]	Yes
CVE-2009-3896	Heap	nginx	283,157	1.10	59	16,821	Yes	[64]	Yes
CVE-2017-9167	Heap	autotrace	75,404	1.01	1,828	2	Yes	No	-
CVE-2018-12326	Heap	Redis	291,275	1.20	8	234	Yes	[65]	Yes
EDB-15705	Heap	ftp	260,986	0.85	19,322	2	Yes	No	-
CVE-2004-1257	Stack	abc2mtex	53,490	0.67	6,319	1	Yes	No	-
CVE-2009-5018	Stack	gif2png	90,738	1.09	1,848	1	Yes	[66]	Yes
CVE-2017-7938	Stack	dmitry	100,186	0.71	4,051	14,402	Yes	No	-
CVE-2018-12327	Stack	ntpq	374,830	1.85	122,740	77,990	Yes	[67]	Yes
CVE-2018-18957	Stack	GOOSE (libiec61850)	65,198	0.71	94	30	Yes	[68]	Yes
CVE-2019-14267	Stack	pdfresurrect	128,427	0.66	83,123	1	Yes	[69]	Yes
* EDB-47254	Stack	abc2mtex	53,490	0.67	6,566	-	Yes	No	-
EDB-46807	Stack	MiniFtp	60,849	0.69	335	107	Yes	No	-
CVE-2006-2025	Integer	GIMP (libtiff)	78,419,067	55.0	3	8	Yes	[70]	Yes
CVE-2007-2645	Integer	exif (libexif)	67,697	0.97	1	7	Yes	[71]	Yes
CVE-2013-2028	Integer	nginx	809,977	2.00	1	25,268	Yes	[72]	Yes
CVE-2017-7529	Integer	nginx	1,049,494	1.10	2	780,404	Yes	[73]	Yes
CVE-2017-9186	Integer	autotrace	75,142	1.00	1	1	Yes	No	-
CVE-2017-9196	Integer	autotrace	74,695	1.03	1	203	Yes	No	-
* CVE-2019-19004	Integer	autotrace	132,302	1.02	1	-	Yes	No	-
CVE-2017-11403	UAF	GraphicsMagick	2,316,152	4.61	38	1	Yes	[74]	Yes
CVE-2017-14103	UAF	GraphicsMagick	2,316,133	4.61	38	1	Yes	[74]	Yes
CVE-2017-9182	UAF	autotrace	132,302	1.02	296	58,058	Yes	No	-
* CVE-2019-17582	UAF	PHP (libzip)	5,980,255	6.40	49	-	Yes	[75]	Yes
CVE-2017-12858	DF	PHP (libzip)	5,980,255	6.40	51	719	Yes	[75]	Yes
* CVE-2019-19005	DF	autotrace	132,302	1.02	57,859	-	Yes	No	-
CVE-2005-0105	FS	typespeed	127,209	0.74	1	1	Yes	[76]	Yes
CVE-2012-0809	FS	sudo	108,442	0.69	1	1	Yes	[77]	Yes
		Average:	**4,568,619**	**5.07**	**11,722**	**36,804**			

* New vulnerability discovered by ARCUS.

[†] Equivalent to applied patch.

The “ΔRoot Cause” column lists how many basic blocks were executed between the state where ARCUS first identifies the vulnerability and its determined root cause point. The numbers vary substantially by class, with heap and stack overflows having distances upwards of 120,000 basic blocks whereas integer overflows and format strings are usually 1.

“ΔAlert” reports the number of blocks between where the runtime monitor flagged the execution and where ARCUS first detected the bug during analysis. In other words, the distance between the monitor alert and the ultimate root cause determined by ARCUS is the sum of “ΔRoot Cause” and “ΔAlert.” Distances vary depending on which monitor was tripped and the overall program complexity. Some executions were not halted until over 700,000 blocks past the bug’s initial symptoms. 0-days found by ARCUS have no reported value since they were not detected by a monitor.

4.3 Consistency to Advisories & Patches

We evaluate the quality of reports for the real-world exploits by manually comparing them against public vulnerability advisories. For example, in CVE-2017-9167, the advisory states that AutoTrace 0.31.1 has a heap-based buffer overflow in the `ReadImage` function defined in `input-bmp.c` on line 337. Accordingly, we expect ARCUS’s root cause report to include the code compiled from this line.

When ARCUS provides a recommendation for extra constraints, we also manually verify that the reported guardian does in fact control the execution of the vulnerable code and that the recommended constraints would prevent the exploit. For example, the ARCUS report for CVE-2018-12327 recommends enforcing at the inner most loop in Figure 1 that a ‘`]`’ character occurs within the first 257 characters of `hname`, as explained in detail in Subsection 3.2. This does prevent the exploit from succeeding, making the report satisfactory.

Some of the evaluated vulnerabilities have already been fixed in newer versions of the targeted programs. In these cases, we use the patch to further verify the quality of ARCUS’s reports by manually confirming that they identify the same code.

Results. The results are shown in the “Located,” “Has Patch,” and “Match” columns of Table 4. All 31 reports correctly identify the exploited vulnerable code. There are patches available at the time of evaluation for 5 of the 8 heap overflows, 4 of the 8 stack overflows, 4 of the 7 integer overflows, 3 of the 4 use after frees, 1 of the 2 double frees, and all 2 format string vulnerabilities. In all but 1 of the 19 official patches available for our tested vulnerabilities, the report generated by ARCUS is consistent with the applied patch. CVE-2004-0597 is a special case where a parent function calls a child using unsafe parameters, causing the child to overflow a heap buffer. ARCUS correctly identifies the vulnerable code, however the developers chose to patch the parent function, whereas ARCUS suggests adding checks inside the child. Both fixes are correct, so this report is satisfactory despite being slightly different from the official patch. 12 of the evaluated vulnerabilities are not patched at the time of evaluation.

4.4 Runtime & Storage Overheads

Dataset & Selection Criteria. To evaluate the performance and storage overheads of ARCUS, we start with the SPEC CPU 2006 benchmark and a storage threshold of 100 GB. We pick this suite because it is commonly used and intentionally designed to stress CPU performance. Since our design requires control flow tracing, CPU intensive tasks are the most costly to trace. I/O tasks by comparison incur significantly less overhead due to blocking, which we demonstrate using Nginx with PHP. Consequently, we consider the SPEC workloads to represent realistic worst case scenarios for ARCUS.

To simulate long-running services and heavy workloads, we stress Nginx and PHP with default settings using ApacheBench (ab) to generate 50,000 requests for files ranging from 100 KB to 100 MB. This experiment also uses a 100 GB storage threshold.

Results. Figure 6 shows the performance and storage overheads of tracing the SPEC workloads without the runtime monitors. The average overhead is 7.21% with a geometric mean of 3.81%, which is consistent with other Intel PT systems [1], [2]. A few workloads have overheads upward of 25%, which is also consistent with prior work and is caused by programs with frequent indirect calls and jumps. A workload yields 110 MB of data on average, which at our chosen storage threshold allows us to store 930 invocations of the program before old data is deleted. In the worst case, we can store 83 invocations.

For the Nginx with PHP stress test, shown in Figure 7, performance overhead is negligible at under 2%. ARCUS generates at most 1.6 MB of data per request, allowing us to store the past 64,000 requests given our 100 GB storage quota. We observe that file size has little influence over storage requirements, with the smallest file producing 1.2 MB of data per request and the largest producing 1.6 MB.

4.5 Case Studies

Discovering Nearby 0-Days. ARCUS discovers that version 1.2.0 of `libzip` has a known vulnerability that can be altered into a new, previously undiscovered, 0-day.[11] Specifically, there is a buggy memory freeing function that maintains a flag in a parent structure to track whether a substructure has already been freed. Calling the freeing function twice on the same structure, without checking the flag, results in a double free (CVE-2017-12858), exploitable via a malformed input.

However, what was not previously known, but uncovered by ARCUS, is that further corrupting the malformed input can trigger a UAF, which has been assigned CVE-2019-17582. Specifically, after freeing the parent structure, invoking the freeing function again can cause it to access the flag that is no longer properly allocated.

Although both bugs reside in the same function, they are distinct — the known CVE double frees the child structure while the new bug inappropriately accesses the parent structure's flag. A developer fixing the prior by more carefully checking the flag will not remediate the latter. ARCUS is able to find this new CVE because it considers all data flows over the executed path.

Vulnerabilities Cascading Into 0-Days. An interesting example in `autotrace` demonstrates how a patch can address one bug, but fail to fix related "downstream" bugs, which gives ARCUS the opportunity to uncover new vulnerabilities. Version 0.31.1 contains a UAF vulnerability exploitable via a malformed input bitmap image header (CVE-2017-9182). Ultimately, ARCUS discovers two additional *downstream* vulnerabilities: an integer overflow (CVE-2019-19004) and a double free (CVE-2019-19005).

They all stem from a lack of input file validation. When the value of the `bits_per_pixel` field of the image header is invalid, after the known UAF, a previously unreported integer overflow can occur as `autotrace` attempts to calculate the number of bytes per row in the `input_bmp_reader` function. ARCUS then discovers an additional double free that releases *the same freed buffer* the UAF accesses. In short, all 3 vulnerabilities are triggered by the same malformed header field, but each resides in a different code block, meaning a developer fixing one may overlook the others.

Vulnerabilities Over Large Distances. Version 0.15 of the program `PDFResurrect` has a buffer overflow vulnerability (CVE-2019-14267) that can be exploited via a malformed PDF to achieve arbitrary code execution. When the function encounters a '`%%EOF`' in the PDF, it scans backwards looking for an '`f`' character, which is supposed to represent the end of '`startxref`'. As it scans, a register representing `pos_count` is incremented. An attacker can create a malformed PDF without a '`startxref`,' causing `pos_count` to exceed 256

[11] Post evaluation, we discovered that this vulnerability had been described in a previous bug report, however it was never issued a CVE ID and so we were unaware of it while evaluating ARCUS. Consequently, we were the first to report it to a CVE authority, resulting in the issuance of CVE-2019-17582.

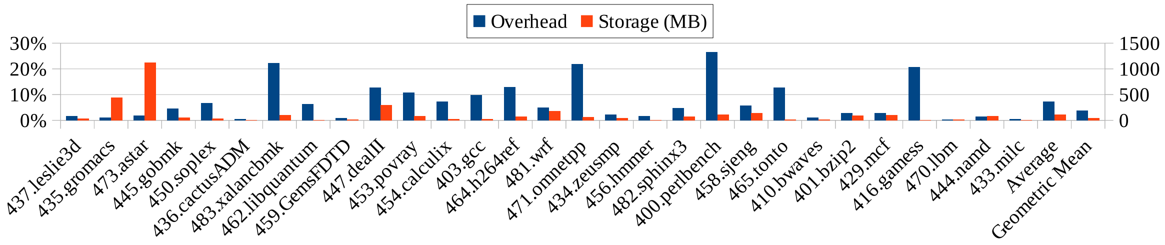


Figure 6: Performance overhead and storage size of tracing the SPEC CPU benchmark. The average overhead is 7.21% and the geometric mean is 3.81%. The average trace size is 110 MB and the geometric mean is 38.2 MB.

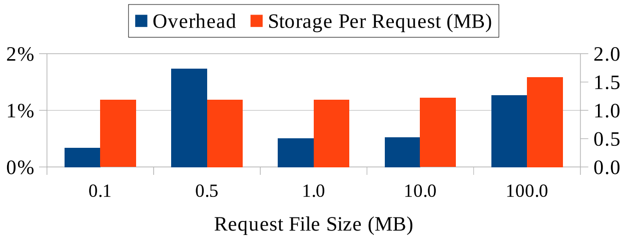


Figure 7: Performance overhead and storage required to trace Nginx. The performance overhead is under 2% and the maximum storage is 1.6 MB per request.

and overflow `buf`. This bug can be exploited to overwrite the stack and achieve arbitrary code execution.

What is interesting about this example is the vulnerable function loads *all* cross references before returning, any one of which could trigger the described overflow. This means thousands of references can be loaded between the corruption point and the return that starts the arbitrary code execution. In our crafted exploit, this distance is over 83,000 basic blocks (see Table 4) and includes almost *17,000* function calls. ARCUS successfully identifies the root cause of the vulnerability despite this distance.

5 Discussion & Limitations

False Negatives & Positives. Prior work enumerates the possible sources of error in symbolic analysis [81], which are not special to ARCUS. ARCUS is a root cause analysis framework invoked in response to an end-host monitor's alert, so it depends on the monitor detecting an attack symptom [82]. As described in Subsection 3.3, some of the modules implemented in ARCUS can incur false negatives.

Only the integer overflow module can yield false positives due to its combination of forward analysis and heuristics. The sole case we have encountered occurs in `libpng`, where an overflowed value is passed to another function, triggering a detection by ARCUS, but then the receiving function performs additional checks, preventing exploitation. Such patterns of checking for overflows in the receiving function (as opposed to the sending) are atypical [58].

Robustness. Recommendations made by ARCUS are based on constraints built from a single execution path, meaning completeness cannot be guaranteed. Human developers are expected to implement the official patch using ARCUS's recommendation as a starting point. Like most solutions that incorporate symbolic analysis, ARCUS is not well suited to building constraints within cryptography procedures, making the current prototype poorly suited for handling bugs within libraries like OpenSSL (e.g., CVE-2010-2939). However, this does not prevent ARCUS from analyzing programs that import such libraries — because the APIs can be modeled — and there are tailored analysis techniques [83] that ARCUS can adopt in future work. Similarly, we do not expect the current ARCUS prototype to perform well on heavily obfuscated binaries or virtual machines (e.g., JVM). The kernel module can trace programs that dynamically generate code, including just-in-time (JIT) compilation, however additional API modeling is required for angr to support web browsers. Conversely, ARCUS already successfully handles some complex programs (e.g., GIMP, 810,000 source lines of C/C++), demonstrating potential for future improvement.

Cross-Platform Support. The current implementation of ARCUS is for x86-64 Linux, but with engineering effort it can support other platforms. Currently, the analysis uses VEX IR semantics, which is machine independent, and angr can lift several hardware architectures. Our "what-if" approach is also machine independent. The integer overflow module leverages some x86-specific semantics to help infer signedness, but it also contains general techniques and can be extended in future work. The memory allocation and format string modules require the semantics for allocation and format string functions (e.g., `printf`, `malloc`). The current prototype supports typical libraries like `libc` and `jemalloc` and prior work proposes techniques for custom functions [84], which can be incorporated in future work.

The largest task is the tracing functionality, which requires an OS module. Although Windows® 10 has an Intel PT driver for tracing applications [85], it is not intended for third-party use and Microsoft® has not released any documentation. While it would be easy for Microsoft to implement ARCUS for Windows, for anyone else, it would require reverse engineering Microsoft's driver [86].

6 Related Work

6.1 Symbolic Execution

The earliest work in symbolic execution demonstrated how executing with symbolic variables can aid in testing and debugging code [87]. As solvers became more efficient, literature emerged for how to use symbolic execution to replay protocols [88] and detect vulnerabilities [89]–[92]. Symbolic execution was also applied to side-channel research [93], firmware analysis [94], correctness of cryptography software [95], emulator testing [96] and automatic binary patching [97].

Much of this work focused on a subset of symbolic analysis called *concolic execution*. Rather than performing pure static analysis, which can get stuck on loops and string parsing, concolic systems leverage real executions for guidance [98]–[100], exploring outwards from the concrete executions to examine as many paths as possible [80], [101]. However, this can lead to state explosion, especially as the analysis deviates further from the concrete execution. This led to hybrid approaches [102], [103], which alternate between fuzzing and symbolic exploration to manage state explosion.

A less explored direction is single path concolic execution, which has proven useful in automatically generating exploits [101], [104], [105] and reverse engineering. The advantage of single path is it sidesteps the issue of state explosion, but it also relies heavily on receiving concrete executions that cover interesting program behaviors. ARCUS distinguishes itself by providing concise root causes using execution traces without needing concrete inputs.

6.2 Root Cause & Crash Dump Analysis

One of the earliest techniques for root cause analysis, delta debugging [106], [107], compares program states between successful and failing inputs to narrow down the set of relevant variables. Another popular approach is to use program slicing to extract only the code that contributes to the failure condition [108]. Delta debugging struggles to generate enough inputs in both classes to be effective while the latter requires tainting or lightweight replay to keep slices small.

Some failure sketching systems handle security bugs like overflows [109], but most focus on race conditions because they are harder to reproduce [110]. Although races have serious security implications, they are not the only class hindering modern programs. There is also work on application layer root cause, including analysis of browser warnings and websites, trace-based pinpointing of insecure keys, and bug finding using written reports, which is orthogonal to ARCUS.

Another direction is crash dump analysis [111], which aims to locate the cause of software crashes. However, while our motivations overlap, our assumptions and scope do not. Crash dump analysis assumes bugs will manifest into crashes, but ARCUS can detect non-crashing exploits. Crash dumps yield partial stack and memory info whereas we have PT traces and snapshots. Data in crash dumps can be corrupt whereas the integrity of PT is protected by the kernel. These factors make our technical challenges significantly different.

7 Conclusion

This work presents ARCUS, a system for performing concise root cause analysis over traces flagged by end-host runtime monitors in production systems. Using a novel "what if" approach, ARCUS automatically pinpoints a concise root cause and recommends new constraints that demonstrably block uncovered vulnerabilities, enabling system administrators to better inform developers about the issue. Leveraging hardware-supported PT, ARCUS decouples the cost of analysis from end-host performance.

We demonstrate that our approach can construct symbolic program states and analyze several classes of serious and prevalent software vulnerabilities. Our evaluation against 27 vulnerabilities and over 9,000 Juliet and RIPE test cases shows ARCUS can automatically identify the root cause of all tested exploits, uncovering 4 new vulnerabilities in the process, with 0 false positives and negatives. ARCUS incurs a 7.21% performance overhead on the SPEC 2006 CPU benchmark and scales to large programs compiled from over 810,000 lines of C/C++ code.

Acknowledgments

We thank the anonymous reviewers for their helpful and informative feedback. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-19-1-2179, N00014-17-1-2895, N00014-15-1-2162, and N00014-18-1-2662, and the Defense Advanced Research Projects Agency (DARPA) under contract HR00112090031. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR or DARPA.

References

[1] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[2] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," in *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017.

[3] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive call-site sensitive control flow integrity," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2019, pp. 95–110.

[4] W. He, S. Das, W. Zhang, and Y. Liu, "Bbb-cfi: Lightweight cfi approach against code-reuse attacks using basic block information," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 1, pp. 1–22, 2020.

[5] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: Cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 941–951.

[6] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.

[7] L. Feng, J. Huang, J. Hu, and A. Reddy, "Fastcfi: Real-time control flow integrity using fpga without code instrumentation," in *International Conference on Runtime Verification*, Springer, 2019, pp. 221–238.

[8] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 577–587.

[9] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 173–184.

[10] B. Niu and G. Tan, "Rockjit: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1317–1328.

[11] S. Forrest, S. Hofmeyr, and A. Somayaji, "The evolution of system-call monitoring," in *2008 Annual Computer Security Applications Conference (ACSAC)*, IEEE, 2008, pp. 418–430.

[12] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," 2010.

[13] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.

[14] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, "Language-independent sandboxing of just-in-time compilation and self-modifying code," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 355–366.

[15] J. A. Kroll, G. Stewart, and A. W. Appel, "Portable software fault isolation," in *2014 IEEE 27th Computer Security Foundations Symposium*, IEEE, 2014, pp. 18–32.

[16] B. Patel, *Intel Releases New Technology Specifications to Protect Against ROP attacks*, https://software.intel.com/content/www/us/en/develop/blogs/intel-release-new-technology-specifications-protect-rop-attacks.html, [Online; accessed 26-June-2020].

[17] *Control Flow Guard*, https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, [Online; accessed 26-June-2020].

[18] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 133–147.

[19] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, IEEE, 2006.

[20] J. Newsome, D. Brumley, D. Song, J. Chamcham, and X. Kovah, "Vulnerability-specific execution filtering for exploit prevention on commodity software.," in *NDSS*, 2006.

[21] A. Slowinska and H. Bos, "The age of data: Pinpointing guilty bytes in polymorphic buffer overflows on heap or stack," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, IEEE, 2007, pp. 487–500.

[22] K. Bhat, E. Van Der Kouwe, H. Bos, and C. Giuffrida, "Probeguard: Mitigating probing attacks through reactive program transformations," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 545–558.

[23] Y. Kwon, B. Saltaformaggio, I. L. Kim, K. H. Lee, X. Zhang, and D. Xu, "A2c: Self destructing exploit executions via input perturbation," in *Network and Distributed Systems Security (NDSS) Symposium 2017*, 2017.

[24] R. Ding, H. Hu, W. Xu, and T. Kim, "Desensitization: Privacy-aware and attack-preserving crash report," in *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[25] F. Capobianco, R. George, K. Huang, T. Jaeger, S. Krishnamurthy, Z. Qian, M. Payer, and P. Yu, "Employing Attack Graphs for Intrusion Detection," in *New Security Paradigms Workshop*, ser. NSPW'19, 2019.

[26] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings 1996 IEEE Symposium on Security and Privacy*, 1996, pp. 120–128. DOI: 10.1109/SECPRI.1996.502675.

[27] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage," in *26th ISOC Network and Distributed System Security Symposium*, ser. NDSS'19, 2019.

[28] X. Han, T. Pasqueir, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," in *27th ISOC Network and Distributed System Security Symposium*, ser. NDSS'20, 2020.

[29] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality.," in *Proceedings of the 12th ISOC Network and Distributed System Security Symposium*, ser. NDSS'05, 2005.

[30] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: Real-time apt detection through correlation of suspicious information flows," in *2019 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA: IEEE Computer Society, 2019.

[31] X. Shu, D. (Yao, N. Ramakrishnan, and T. Jaeger, "Long-span program behavior modeling and attack detection," *ACM Transactions on Privacy and Security*, vol. 20, 2017.

[32] A. Wespi, M. Dacier, and H. Debar, "Intrusion detection using variable-length audit trail patterns," in *Recent Advances in Intrusion Detection*, Springer, 2000, pp. 110–129.

[33] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344)*, 1999, pp. 133–145.

[34] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Zhen, W. Cheng, C. A. Gunter, and H. chen, "You are what you do: Hunting stealthy malware via data provenance analysis," in *27th ISOC Network and Distributed System Security Symposium*, ser. NDSS'20, 2020.

[35] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 467–478.

[36] M. Bellare and B. Yee, "Forward integrity for secure audit logs," Computer Science and Engineering Department, University of California at San Diego, Tech. Rep., 1997.

[37] J. E. Holt, "Logcrypt: Forward security and public verification for secure audit logs," in *Proceedings of the Australasian Information Security Workshop (AISW-NetSec)*, 2006.

[38] R. Paccagnella, K. Liao, D. (Tian, and A. Bates, "Logging to the danger zone: Race condition attacks and defenses on system audit frameworks," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS'20, 2020.

[39] B. Schneier and J. Kelsey, "Cryptographic support for secure logs on untrusted machines.," in *Proceedings of the USENIX Security Symposium (USENIX)*, 1998.

[40] D. Ma and G. Tsudik, "A new approach to secure logging," *ACM Transactions on Storage (TOS)*, vol. 5, no. 1, 2009.

[41] A. A. Yavuz and P. Ning, "Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009.

[42] A. A. Yavuz, P. Ning, and M. K. Reiter, "Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging," in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[43] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs.," in *OSDI*, vol. 8, 2008, pp. 267–280.

[44] I. Ahmed, N. Mohan, and C. Jensen, "The impact of automatic crash reports on bug triaging and development in mozilla," in *Proceedings of The International Symposium on Open Collaboration*, 2014, pp. 1–8.

[45] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, and A. Kaseorg, "Security impact ratings considered harmful," *arXiv preprint arXiv:0904.4058*, 2009.

[46] P. J. Guo and D. R. Engler, "Linux kernel developer responses to static analysis bug reports.," in *USENIX Annual Technical Conference*, 2009, pp. 285–292.

[47] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, Apr. 2010.

[48] S. Ren, L. Tan, C. Li, Z. Xiao, and W. Song, "Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, Jun. 2016.

[49] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008, pp. 1–14.

[50] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand inter-process information flow tracking," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[51] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage.," in *NDSS*, 2019.

[52] M. E. Aminanto, L. Zhu, T. Ban, R. Isawa, T. Takahashi, and D. Inoue, "Automated threat-alert screening for battling alert fatigue with temporal isolation forest," in *2019 17th International Conference on Privacy, Security and Trust (PST)*, IEEE, 2019, pp. 1–3.

[53] S. McElwee, J. Heaton, J. Fraley, and J. Cannady, "Deep learning for prioritizing and responding to intrusion detection alerts," in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, IEEE, 2017, pp. 1–5.

[54] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[55] C. Yagemann, S. Sultana, L. Chen, and W. Lee, "Barnum: Detecting document malware via control flow anomalies in hardware traces," in *Proceedings of the 25th Information Security Conference (ISC)*, New York, NY, USA, 2019.

[56] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[57] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, 2010, pp. 1–1.

[58] T. Wang, C. Song, and W. Lee, "Diagnosis and emergency patch generation for integer overflow exploits," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2014, pp. 255–275.

[59] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "Ripe: Runtime intrusion prevention evaluator," in *In Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, ACM, 2011.

[60] P. E. Black and P. E. Black, *Juliet 1.3 Test Suite: Changes From 1.2*. US Department of Commerce, National Institute of Standards and Technology, 2018.

[61] *CVE-2004-0597 Patch*, https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2004-0597#patch, [Online; accessed 25-October-2019].

[62] *CVE-2004-1288 Patch*, https://pastebin.com/raw/fsFkspFF, [Online; accessed 25-October-2019].

[63] *Red Hat Bugzilla – Attachment 360889 Details for Bug 523105*, https://bugzilla.redhat.com/attachment.cgi?id=360889&action=diff, [Online; accessed 07-January-2020].

[64] *Debian Bug report logs - #552035*, https://bugs.debian.org/cgi-bin/bugreport.cgi?att=1;bug=552035;filename=diff;msg=16, [Online; accessed 10-January-2020].

[65] *Commit 3f730d50*, https://github.com/antirez/redis/commit/3f730d50, [Online; accessed 16-January-2020].

[66] gif2png, *Command Line Buffer Overflow*, https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=550978#50, [Online; accessed 25-October-2019], 2009.

[67] ntp, *Stack-based buffer overflow in ntpq and ntpdc allows denial of service or code execution*, https://bugzilla.redhat.com/show_bug.cgi?id=1593580, [Online; accessed 25-October-2019], 2018.

[68] *Commit 5470551c*, https://github.com/mz-automation/libiec61850/commit/5470551c, [Online; accessed 09-April-2020].

[69] pdfresurrect, *Prevent a buffer overflow in possibly corrupt PDFs*, https://github.com/enferex/pdfresurrect/commit/3f811dbc, [Online; accessed 25-October-2019], 2019.

[70] libtiff, *Multiple libtiff Issues*, https://bugzilla.redhat.com/attachment.cgi?id=128255&action=diff, [Online; accessed 25-October-2019], 2006.

[71] EXIF Tag Parsing Library, *#70 SERIOUS SECURITY BUG IN EXIF_DATA_LOAD_DATA_ENTRY()*, https://sourceforge.net/p/libexif/bugs/70/, [Online; accessed 25-October-2019], 2007.

[72] *patch.2013.chunked.txt*, https://nginx.org/download/patch.2013.chunked.txt, [Online; accessed 16-January-2020].

[73] *patch.2017.ranges.txt*, https://nginx.org/download/patch.2017.ranges.txt, [Online; accessed 16-January-2020].

[74] GraphicsMagick, *Attempt to Fix Issue 440*, http://hg.code.sf.net/p/graphicsmagick/code/rev/98721124e51f, [Online; accessed 25-October-2019], 2017.

[75] libzip, *Fix double free*, https://github.com/nih-at/libzip/commit/9179b796, [Online; accessed 25-October-2019], 2017.

[76] *CVE-2005-0105 Patch*, https://pastebin.com/raw/GHm1k1Rk, [Online; accessed 25-October-2019].

[77] sudo, *Format String Vulnerability*, https://bugs.gentoo.org/401533, [Online; accessed 25-October-2019], 2012.

[78] *LinuxFlaw*, https://github.com/VulnReproduction/LinuxFlaw, [Online; accessed 06-January-2020].

[79] *Exploit Database*, https://www.exploit-db.com/, [Online; accessed 06-January-2020].

[80] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22nd USENIX Security Symposium*, 2013, pp. 49–64.

[81] D. Zhang, D. Liu, Y. Lei, D. Kung, C. Csallner, and W. Wang, "Detecting vulnerabilities in c programs using trace-based testing," in *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010.

[82] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1675–1689.

[83] S. Y. Chau, M. Yahyazadeh, O. Chowdhury, A. Kate, and N. Li, "Analyzing semantic correctness with symbolic execution: A case study on pkcs# 1 v1. 5 signature verification.," in *NDSS*, 2019.

[84] X. Chen, A. Slowinska, and H. Bos, "Who allocated my memory? detecting custom memory allocators in c binaries," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, IEEE, 2013, pp. 22–31.

[85] *WindowsIntelPT*, https://github.com/intelpt/WindowsIntelPT, [Online; accessed 12-June-2020].

[86] *winipt*, https://github.com/ionescu007/winipt, [Online; accessed 12-June-2020].

[87] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[88] J. Newsome, D. Brumley, J. Franklin, and D. Song, "Replayer: Automatic protocol replay by binary analysis," in *Proceedings of the 13th ACM conference on Computer and communications security*, ACM, 2006, pp. 311–321.

[89] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, Springer, 2008, pp. 1–25.

[90] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ACM, 2009, pp. 225–236.

[91] D. A. Molnar and D. Wagner, "Catchconv: Symbolic execution and run-time type inference for integer conversion errors," *UC Berkeley EECS*, 2007.

[92] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing.," in *NDSS*, Citeseer, vol. 8, 2008, pp. 151–166.

[93] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," in *CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation*, IEEE, 2019.

[94] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.," in *NDSS*, 2015.

[95] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li, "Symcerts: Practical symbolic execution for exposing noncompliance in x. 509 certificate validation implementations," in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 503–520.

[96] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 40, 2012, pp. 337–348.

[97] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, *et al.*, "Mechanical phish: Resilient autonomous hacking," *IEEE Security & Privacy*, vol. 16, no. 2, pp. 12–22, 2018.

[98] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation.," in *NDSS*, 2011.

[99] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 30, 2005, pp. 263–272.

[100] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[101] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 33rd Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[102] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution.," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[103] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.

[104] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Automatic exploit generation," Carnegie Mellon University, 2018.

[105] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2018, pp. 1914–1927.

[106] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[107] J.-D. Choi and A. Zeller, "Isolating failure-inducing thread schedules," in *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 27, 2002, pp. 210–220.

[108] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve, "Using likely invariants for automated software fault localization," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 41, 2013, pp. 139–152.

[109] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "Pres: Probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ACM, 2009, pp. 177–192.

[110] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: A technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[111] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "Postmortem program analysis with hardware-enhanced post-crash artifacts," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 17–32.