

Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis

Carter Yagemann

Georgia Institute of Technology
Atlanta, Georgia, USA

Brendan Saltaformaggio

Georgia Institute of Technology
Atlanta, Georgia, USA

Simon P. Chung

Georgia Institute of Technology
Atlanta, Georgia, USA

Wenke Lee

Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

The increasing cost of successful cyberattacks has caused a mindset shift, whereby defenders now employ proactive defenses, namely software bug hunting, alongside existing reactive measures (firewalls, IDS, IPS) to protect systems. Unfortunately the path from hunting bugs to deploying patches remains laborious and expensive, requires human expertise, and still misses serious memory corruptions. Motivated by these challenges, we propose bug hunting using symbolically reconstructed states based on execution traces to achieve better detection and root cause analysis of overflow, use-after-free, double free, and format string bugs across user programs and their imported libraries. We discover that with the right use of widely available hardware processor tracing and partial memory snapshots, powerful symbolic analysis can be used on real-world programs while managing path explosion. Better yet, data can be captured from production deployments of live software on end-host systems transparently, aiding in the analysis of user clients and long-running programs like web servers.

We implement a prototype of our design, Bunkerbuster, for Linux and evaluate it on 15 programs, where it finds 39 instances of our target bug classes, 8 of which have never before been reported and have led to 1 EDB and 3 CVE IDs being issued. These 0-days were patched by developers using Bunkerbuster's reports, independently validating their usefulness. In a side-by-side comparison, our system uncovers 8 bugs missed by AFL and QSYM, and correctly classifies 4 that were previously detected, but mislabeled by AddressSanitizer. Our prototype accomplishes this with 7.21% recording overhead.

CCS CONCEPTS

• **Security and privacy** → **Systems security; Software and application security.**

KEYWORDS

symbolic analysis, processor tracing, bug hunting

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8454-4/21/11...\$15.00

<https://doi.org/10.1145/3460120.3485363>

ACM Reference Format:

Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea*. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3460120.3485363>

1 INTRODUCTION

As pressure on companies to swiftly identify and remediate system vulnerabilities has increased [47], corporations have adopted *bug hunting* strategies. They *proactively* search for and remediate problems in their adopted software *before* adversaries can exploit them in an attack [70]. Unfortunately, the path from corporate bug hunting to developer software patch is cumbersome and laborious, leaving less-equipped companies vulnerable.

Human bug hunters, lacking good inputs to test programs, rely on fuzz testing (fuzzing) [23, 72, 85, 86, 110] to brute force test cases, starting from seeds provided by the developers (e.g., regression tests) or scraped from public databases (e.g., ImageNet [33]) that offer limited coverage. Such tools often require manually written scaffolding code to reach deep libraries or APIs [10, 52, 56] and rely on crashes to signal buggy behavior [11, 93, 99, 111], which is not always reliable [34, 37, 44]. The process is further complicated by binaries that lack source code, requiring bug hunters to engage in extensive reverse engineering [36, 39, 43].

Worse still, the bug hunter then needs to share their findings with the software's developers. Crashes can corrupt artifacts [14, 21, 38, 49, 50, 53, 65, 71, 73, 79, 89, 96, 102–105, 108, 109] and bugs can be difficult to reproduce due to environment differences. Capturing stack traces or re-executing the crashing input with instrumentation [57, 61, 91, 97] offers some insights, but as we discover in an in-depth case study, the results can be incomplete, hindering triage. Prior work shows that developers consistently undervalue or ignore issues they do not understand [9, 46], but without their aid, the only other remediation choices are incomplete stopgaps like input filters [20, 30, 76, 97] or selective function hardening [15], which incur significant overhead [67].

However, we observe that software testing need not occur in a vacuum. Namely, companies already have employees constantly using the software in question, and their *real-world usage already drives the program into deep behaviors within realistic environments*. The data to automate bug hunting and reporting is already within their reach, so why are they not using it?

We hypothesize that the disconnect that occurs is due to the traditional definition of “seed as program input” being insufficient. While program inputs are easy to collect, they offer little insight into how to build scaffolding, how to get from sound program states to buggy ones, and how to explain those bugs in a meaningful way. Instead, we hypothesize that *control flow traces* are the better seeds for automating bug hunting and reporting because they can reveal the solution to all the above problems while still being efficient enough to collect from real user environments.

To demonstrate this, we first propose how to segment control flow traces and save sequential memory snapshots to guide symbolic analysis through code where it is otherwise susceptible to path explosion [13, 88]. We hypothesize that this *control oriented* record and replay of user sessions is suitable for discovering serious classes of memory corruption, such as those arising from overflows, use-after-free (UAF), double free (DF), and format string (FS) bugs. Better still, thanks to the prevalence of hardware assisted processor tracing¹ (PT), production systems can securely capture traces with user transparency and tolerable overhead.

Notice that while prior work has demonstrated the value of snapshots for bug hunting [58], they did not combine them with traces. Without the accompanying segmented control flow traces leveraged in our design, such systems are still susceptible to path explosion due to loops and string manipulation, limiting their scalability in real-world settings.

However, collecting a corpus of new seeds is only half the battle. To reach new buggy program states, we also propose a technique to selectively symbolize predicate data, based on the recorded control flow traces, to facilitate *constrained* exploration that *prioritizes* certain paths while managing path explosion. To inspect deep API calls, we propose an analysis to automatically recover parameter prototypes, eliminating the need for human analysts to implement scaffolding. To find bugs from benign recordings, we employ bug-class-specific search strategies and detection techniques that check uncovered states for symbolic indicators of buggy behavior.

The above technical contribution also brings an additional benefit to our design, which is that the same symbolic constraints can also be used to perform symbolic root cause analysis [106]. This recently proposed technique for localizing memory corrupting bugs has only been demonstrated in single path symbolic analysis, starting from the program entry point, limiting its possible applications. Our design shows how it can be used in a multi-path setting, starting from the main program entry point *or* entry points to imported library APIs, increasing its applicability.

We implement our design as a Linux prototype, named Bunkerbuster, and evaluate it on 15 programs, some of which contain binaries compiled from over 810,000 lines of C/C++ code, invoking 1,710 imported functions and producing traces 19,392,602 basic blocks long, on average. Bunkerbuster successfully uncovers 39 bugs, of which 8 are newly discovered by our approach. 1 EDB and 3 CVE IDs have been issued and patched by developers, using Bunkerbuster’s reports to independently verifying their novelty.² In a side-by-side comparison, Bunkerbuster finds 8 bugs missed by AFL and QSYM, and correctly classifies 4 that AddressSanitizer

¹Available in Intel®, AMD®, and ARM® processors.

²We report all bugs to developers, MITRE, and Offensive Security for responsible disclosure.

```

/* coders/png.c */
ReadMNGImage () {
5129:  previous = image; // heap object
5130:  mng_info->image = image;
5138:  ReadOneJNGImage(mng_info); // 1st free
5143:  DestroyImageList(previous); // 2nd free
}

/* coders/png.c */
ReadOneJNGImage(MngInfo *mng_info) {
3126:  DestroyImageList(mng_info->image);
}

/* magick/list.c */
DestroyImageList(Image* images) {
239:  DestroyImage(images); // calls free
}

```

Figure 1: Source code pertaining to CVE-2017-11403 in GraphicsMagick, summarized. ReadMNGImage calls ReadOneJNGImage without realizing that it may free image, making Line 5,143 a DF bug for some paths.

mislabelled. Our prototype accomplishes this with 7.21% recording overhead and manageable storage requirements. We have open sourced our prototype and data to facilitate future work.³

2 OVERVIEW

Bunkerbuster’s analysis replaces the laborious process of *proactively* hunting for and reporting software bugs in enterprise networks. Bug hunting should not be confused with intrusion detection (IDS) or prevention (IPS), which requires *reacting* swiftly to ongoing attacks. In place of a human security expert creating a testbed to fuzz programs or library APIs, Bunkerbuster gathers data from end-hosts using a kernel driver, cleverly inferring input structures and segmenting traces to achieve offline binary symbolic execution.

2.1 Real-World Example

To show how Bunkerbuster benefits a bug hunter tasked with finding problems in software, consider the following example based on CVE-2017-11403, a UAF vulnerability found in GraphicsMagick. For clarity, we will explain this example using the source code shown in Figure 1, however the real analysis is on binaries. In this instance, the function ReadMNGImage always frees the heap object `image` before returning, but what it does not account for is that a child function it invokes, ReadOneJNGImage, can also free `image` after a certain error, causing a DF.

Suppose that the bug hunter, having heard about all the recent vulnerabilities found in image processing libraries, wants to analyze a program his employees are using that imports the GraphicsMagick library. Unfortunately, he is not familiar with obscure image formats like MNG, so building fuzzer scaffolding for all of GraphicsMagick’s APIs would be tedious, and fuzzing the entire program from startup would be inefficient due to its complexity.

³<https://github.com/carter-yagemann/arcus>

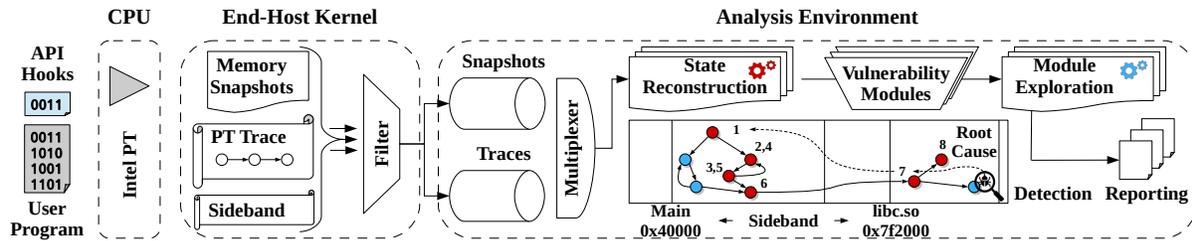


Figure 2: Bunkerbuster architecture. End-hosts with PT-enabled kernel drivers collect and filter traces of the target program, forwarding novel segments to the analysis environment. Symbolic states are reconstructed and then expanded by exploration plugins. When a bug is detected, symbolic root cause analysis pinpoints the source and produces a report.

Instead, he gives the name of the target program to the Bunkerbuster analysis system, which in turn forwards it to all the end-hosts with the Bunkerbuster kernel driver installed, as shown in Figure 2. These systems observe the processes being created locally and anytime the target program starts, they configure PT for recording. As the data is collected at the end-host, it locally segments the trace at calls to imported library functions and hashes them on-the-fly. Each hash is checked against a filter, and if the segment is novel, it is forwarded for analysis.

Back at the analysis system, Bunkerbuster uses the incoming traces along with symbolic execution to reconstruct symbolic states for each executed basic block. Since the conditions leading to CVE-2017-11403 are rare, these segments do not directly reach the DF bug, but some contain invocations of the vulnerable function. Using its search plugins containing bug-class-specific exploration strategies, Bunkerbuster symbolically expands the set of reconstructed states, yielding additional states within the same function, including the one containing the CVE. When Bunkerbuster checks them for memory corruption, it finds the state containing the DF. It then switches to localizing a concise root cause. Bunkerbuster compares the constraints leading to this buggy state against others sharing the same predecessor guardian (i.e., conditional check) and determines the difference that makes the DF reachable. It then traces this back through the predecessor states, pinpointing the error checking branch. The end result is a concise, human-readable report, identifying the site of the first and second frees, and the input error check in GraphicsMagick that caused the DF.

Notice that if no end-user ever loads an MNG image, the analysis will not find this DF because the traces will not have any invocations of the vulnerable function to reference. However, code that is never invoked is a prime candidate for debloating [51, 66, 83, 84, 92], which is outside the scope of this work. Conversely, Bunkerbuster will cover all the code used by monitored users.

2.2 Goals & Assumptions

We focus on discovering and localizing overflow, UAF, DF, and FS bugs within unobfuscated, benign Linux programs without access to source code or debug symbols. The limitations imposed by this scope are discussed further in Section 5.

We assume that the end-hosts contain PT-enabled CPUs, which also form our trusted computing base (TCB). PT is a hardware feature that writes directly to physical memory, bypassing all CPU caches, and is only configurable in the privileged CPU

mode, making it a trusted platform in numerous security systems [31, 32, 40, 62, 107]. We expect collected data to encode benign behaviors, motivating the need for bug hunting. In the event that an end-host captures malicious activity, detection becomes easier. We envision our system being deployed on enterprise computers and servers, leaving mobile and embedded devices for future work.

To recover the structure of inputs to APIs as accurately as possible while covering the diverse range of possible use cases, we consider two scenarios. The first targets open source C/C++ libraries, where we assume access to stub code or source headers that define the API. This is a necessary part of any public release to allow other developers to integrate their systems with the API. For all other cases, we assume the most conservative scenario where only the binary is available.

3 DESIGN & IMPLEMENTATION

In this section, we elaborate on the steps in Bunkerbuster’s recording and analysis, initially presented at a high level in Subsection 2.1. Stepping through the workflow sequentially, Subsection 3.1 describes how the end-hosts record and filter the PT traces that the analysis uses to recover valid program execution paths. Subsection 3.2 then describes how memory snapshots are taken at the end-host and how the analysis selectively symbolizes them to bootstrap symbolic execution. Given a symbolized snapshot as a starting state and a matching trace segment, Subsection 3.3 describes how to recover symbolic representations of all the intermediate program states along the traced path at basic block granularity.

With a linear sequence of symbolic states for the recorded path constructed, we then describe how to explore additional paths, prioritized using search strategies based on our domain knowledge of our target bug classes. We also describe how Bunkerbuster uses the symbolic constraints for the states to detect and then localize bugs. Since our techniques are bug-class-specific, we split our description between UAF/DF, which arise from *temporal* memory safety violations, and overflow/FS, which arise from *spatial* memory safety violations, in Subsections 3.4 and 3.5, respectively.

3.1 Capturing & Filtering Traces

One technical challenge Bunkerbuster has to overcome is how to efficiently, securely, and transparently record user sessions. To this end, we center our design around PT, and then propose a novel way of hashing recorded segments so redundant ones can be discarded.

However, before explaining filtering, it is important to understand what PT is and how Bunkerbuster uses it. For brevity, we will focus on Intel’s implementation of PT, however similar features exist in processors made by ARM, AMD, and others.

Intel PT records traces of user space execution directly to physical memory, where it can then be forwarded by a kernel driver to persistent storage or remote endpoints. Its recording can be restricted to a particular process at the hardware level using a configuration register that accepts a CR3 value representing the process’ page table address.

Traces consist of a stream of packets, each recording the outcome of a branching instruction, indirect call/jump, return, or interrupt. Binary branches are recorded as a single taken-not-taken (TNT) bit, whereas other events yield a target instruction pointer (TIP). To decode the trace into an instruction sequence, the decoder also needs additional *side-band data* about the traced process’ memory space and thread scheduling, which we describe next.

First, the decoder needs the process’ executable pages in order to recover instructions. Bunkerbuster’s kernel driver handles this by hooking relevant system calls (e.g., `mmap`, `mprotect`) and recording memory pages alongside the PT trace. Bunkerbuster can then linearly disassemble the memory, starting at the program’s entry point and consulting the next PT packet whenever a branch is encountered, to recover every executed instruction.

Second, in order to distinguish threads that share the same page table (CR3 value), the kernel driver also hooks context switches to record when threads are swapped in and out of CPU cores. The driver also hooks the `fork` and `exec` system calls so it can detect and trace child processes created by the target program.

Trace Filtering. Unlike prior PT systems, Bunkerbuster has to account for the fact that users and services may engage in repetitive tasks, yielding partially redundant execution traces. To address this, our driver quickly hashes trace segments on-the-fly and compares them against a global map, discarding ones that have already been observed, using the following algorithm:

$$(u, v) \in T : u \ll 1 \oplus v \bmod S \quad (1)$$

where u and v are virtual address offsets, relative to their object bases to account for ASLR, recovered from trace T . The result is a bit offset within a map of size S bits corresponding to the edge (u, v) . The global map is initialized with all bits set to 0 and then as edges are decoded from the PT trace, their corresponding bits are set to 1. If a trace segment adds any novel bits to the global map, it is forwarded for analysis, otherwise it is discarded.

3.2 Symbolizing Memory Snapshots

Alongside the data described in Subsection 3.1, the end-host driver also records snapshots of register values and memory that will serve as starting states for symbolic execution. Specifically, when the program is loaded at runtime, the driver hooks the program’s main entry point and any entrances to imported APIs (i.e., library functions) by placing traps in the process’ procedure linkage table (PLT). Once captured, Bunkerbuster symbolizes the input data, which for the main entry point is the program’s input arguments and for APIs are the called function’s parameters. This data is replaced with unconstrained symbolic variables, enabling Bunkerbuster to

```
000000000001142 <foobar>:
...
114a: mov    %rdi, -0x18(%rbp)  u: {rdi}
114e: mov    %esi, -0x1c(%rbp)  u: {rdi, esi}
1151: mov    %rdx, -0x28(%rbp)  u: {rdi, esi, rdx}
...
1164: mov    -0x18(%rbp), %rax
1168: add   %rdx, %rax
116b: movzbl (%rax), %eax
116e: movsbl %al, %eax        u: {rdi[s8], esi, rdx}
...
1183: mov    -0x28(%rbp), %rax
...
118c: callq  *%rax            u: {rdi[s8], esi, rdx[c]}
```

Figure 3: Binary-only scenario, with color added for clarity. The boxes show the usage of non-clobbered values. The first snippet reveals foobar has 3 arguments, the next reveals that the RDI argument is a char pointer (denoted [s8]), and the last reveals RDX is a code pointer ([c]).

reason about all possible input values to the program and imported APIs. For this reason, each trap only needs to be used once, and is then removed, minimizing runtime overhead. This also allows Bunkerbuster to analyze snapshots (and their corresponding trace segments) in any order because there are no prior constraints.

Generally speaking, under-constrained symbolic execution can result in false positive detections, i.e., bugs that cannot actually be reached in real executions. However, because we are careful to only snapshot the entry points to the program and its imported libraries, Bunkerbuster’s results do not have this issue. Bugs found using snapshots of the program’s entry point will be inherently reachable, and of relevance to the program’s developers. Conversely, for API snapshots, so long as Bunkerbuster halts its analysis at the return from the called function,⁴ any discovered bugs *may not* be reachable within the context of the program that was recorded, but *may* be reachable by other programs that also import the same library, making the results relevant to library developers. In this way, Bunkerbuster decomposes long traces into smaller segments, simplifying the symbolic execution.

One small caveat we discovered while designing Bunkerbuster is that while most inputs within snapshots should be symbolized, *code pointers passed to APIs should not*. The reason is that some APIs are designed to accept code pointers, which may serve as callback functions, helper functions, and more. If these are replaced with unconstrained symbolic variables, then their use will be difficult to distinguish from control flow hijacking, despite being intended behavior. The reason why will become clearer in Subsection 3.5, which describes how Bunkerbuster detects overflow bugs.

Whereas program arguments adhere to a fixed memory layout, as specified by the operating system, the locations and types of API arguments has to be recovered by Bunkerbuster. Recall from Subsection 2.2 that we aim to handle both public and private APIs. Consequently, we propose two approaches for inferring and symbolizing the input arguments, one based on parsing C/C++ headers and the other based on binary-only analysis.

⁴Analysis beyond this point can yield false positives because the returned value is under-constrained.

Source-Based Inference. When source headers are available, we use a C/C++ parser to read the API’s function prototype into an abstract syntax tree (AST), terminating with basic data types of known size (e.g., int, void pointer). All non-pointer types are treated as data. For pointers, if the type is a function prototype, then it is a code pointer. Similarly, pointers to basic data types are data. However, it is ambiguous when the type is void, which could point to data or code. In such cases, the parser assumes the pointer points to code to remain conservative. The result is a data structure defining the offset, size, and type of each element for each argument. This is then combined with the calling convention for the architecture being analyzed to pinpoint these elements in registers and memory. Notice that because libraries are shared between programs, factors like padding are treated consistently across systems and is easy to account for. When data pointers point to buffers of arbitrary length, they are replaced with new large buffers of unconstrained symbolic bytes to test for overflows.

Binary-Based Inference. When headers are unavailable, our analysis leverages the recorded trace, shown with a concrete example in Figure 3. Bunkerbuster steps through the traced basic blocks in order and tracks where registers and stack values are used in operations versus being clobbered by writes. If a non-clobbered value is used, it is likely an argument. The type is inferred based on how the loaded value is manipulated. If it appears in a call, it is treated as a code pointer. If it is used in subsequent loads, it is a data pointer. Otherwise it is treated as a basic data type. It is possible for this approach to miss a parameter if it is never used, however we did not observe this in our evaluation.

During implementation, we tested the robustness of this approach by comparing its outputs against those of the source-based technique and verifying that they match. We include a breakdown of the tested libraries in Table 5 of the Appendix.

3.3 Symbolic State Reconstruction

Once Bunkerbuster has a symbolized snapshot for a starting state (Subsection 3.2), and a corresponding trace segment (Subsection 3.1), it then needs to recover the intermediate program states that cover the recorded execution path. Notice that Bunkerbuster cannot simply take more snapshots because doing so comes at a performance cost, so instead our solution is to use symbolic execution to recover the missing states. As an added benefit, this will also enable Bunkerbuster to consider states beyond what was concretely executed, potentially finding additional bugs.

To perform the reconstruction, each instruction is emulated and constraints are added to the programs state to encode all possible data that can reach the current point in the execution. When a branching instruction is encountered, a *satisfiability modulo theories* (SMT) solver evaluates the accumulated constraints to yield reachable successor states. However, Bunkerbuster initially focuses on only recovering the path that was recorded, so it only keeps the successor that matches the next address in the trace. In this way, there is only 1 active state per step.

CPU Architecture-Specific Considerations. Although following a linear sequence of executed addresses is conceptually intuitive, in practice real-world encoding schemes can introduce ambiguities

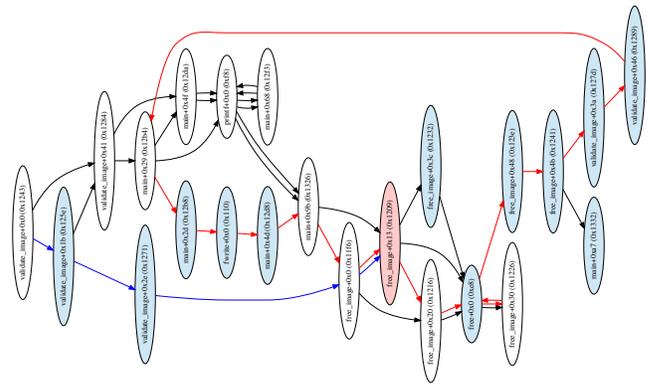


Figure 4: CFG created by the UAF module for a real-world case (subgraph shown for brevity). Black edges are the path traced by PT and blue nodes are states the module discovered. The blue edges show a discovered path leading to a free, followed by the red path leading to a UAF bug (red node).

that must be resolved carefully. One prevalent case occurs in processors supporting extended instruction sets (ISAs), such as IA64 and AMD64. Among the added instructions are complex operations like Intel’s “repeat” instructions, which allow compilers to implement an entire loop in one instruction.⁵ When executing concrete memory in a real processor, these instructions are deterministic, so Intel PT ignores them. However, in symbolic analysis, two successor states become reachable if symbolic memory is accessed: one that completes the instruction and another that continues its iterating. Since the trace offers no guidance, our solution is to “iterate” on the repeat instruction as many times as possible, given the symbolic constraints, because this is most likely to reveal an overflow bug. Once the analysis must advance past the complex instruction, it synchronizes back to the trace and continues.

3.4 Use-After-Free & Double Free Bugs

The UAF module (also covering DF) relies on a value set analysis (VSA) over the symbolic states. However, unlike a typical VSA that tracks the *concrete* pointers to allocated and freed memory buffers, Bunkerbuster’s VSA is performed using *symbolic* pointers, constrained by the symbolic execution to encode all possible values at the current program state. This carries several advantages. For example, in the evaluation presented in Subsection 4.3, we encounter a case where AddressSanitizer, having access to only a single concrete input provided by a fuzzer, concluded that a pointer passed to free could cause an invalid free, since the pointer’s value was an address that was not allocated. However Bunkerbuster, using the symbolic representation of that same pointer, detected that there were other satisfiable values for it, some of which corresponded to addresses that *were* allocated, revealing the bug to really be a UAF.

Detection. To perform the VSA, we assume knowledge of the syntax of memory management functions in advance, which is easily achievable in practice because most programs rely on a few

⁵strlen can be implemented in IA64 using a single repnz scasd instruction.

standard implementations. Even when a wrapper is placed around memory management functions for portability across systems, we find that Bunkerbuster can track the underlying standard library while disregarding the wrapper. In the case of the real-world programs in our evaluation dataset, they all rely on either `libc` or `jemalloc`. There are also algorithms to automatically detect and infer memory management functions [26], which can be incorporated in future work, but are not implemented in our initial prototype.

When the program calls into an allocation function, Bunkerbuster records the locations of the pointer, the allocated buffer, and its size, in an *allocated* set. If the size is symbolic, Bunkerbuster evaluates it to its maximum satisfiable value. When a pointer is passed to a `free` function, Bunkerbuster evaluates the symbolic constraints to determine which buffer is being referenced and moves it into a *freed* set. Notice that the referenced buffer could be one that is already freed, in which case a DF bug is detected. Similarly, Bunkerbuster checks any dereferenced pointers in each discovered state, and if one can point to a freed buffer, it is a UAF bug.

Search. With symbolic buffer and pointer metadata recovered via VSA, Bunkerbuster’s search strategy first recovers function boundaries, which are determined based on the calls and returns contained within the trace, and then labels which functions manipulate heap based on the collected pointer metadata. The implementation of the algorithm to recover all the accessed memory addresses for a basic block is provided in the Appendix as Algorithm 1. Bunkerbuster then searches these functions for additional states using depth-first-search to see if they can cause a UAF or DF. By sticking only to functions reached during tracing, Bunkerbuster can avoid path explosion by returning to any of the traced states reconstructed in Subsection 3.3.

Figure 4 shows a partial control flow graph (CFG) for a UAF bug found with this strategy. The initial states from the trace are shown in white, connected by black edges. Nearby states found during exploration are shown in blue, revealing the blue path to a `free`. Further exploration of this and other traced functions then reveals the red path leading to a UAF.

Root Cause. Once detected, the symbolic root cause report prepared for the developers contains the basic block that allocated the accessed buffer, the one that freed it, and the one that performed the buggy access. To propose a preliminary patch, the module constructs a control dependency graph (CDG) over the path leading to the UAF, revealing all the conditional branches the violating basic block’s reachability depends on. The branch nearest to the violator is selected (based on shortest path) and the state for the alternate branch (which did *not* cause a bug) is checked for its constraints. If these constraints contradict the UAF state, this becomes the preliminary patch, otherwise the report advises the developers to place a new guard condition before the violating basic block.

3.5 Overflow & Format String Bugs

Detection. Bunkerbuster’s overflow detection module focuses on bugs that can manifest into control flow hijacking, taking advantage of the fact that all external input data is symbolized in the starting memory snapshot (Subsection 3.2). Consequently, if the program counter for a state ever becomes symbolic due to one of

these variables, this means external input can directly control the execution of the code via crafted inputs, which is a serious vulnerability. Notice that symbolic constraints are already propagated by the symbolic execution, so detection is performed by querying the SMT solver to check whether the program counter is symbolic (i.e., has more than 1 satisfiable value). If it is, an overflow has occurred.

Search. Bunkerbuster searches for overflows by identifying all the loops that appear in the trace, which is accomplished by transforming the linear execution into a CFG and then using a depth-first search to find all the backward edges in the graph.⁶ Once identified, the module’s search strategy is to *stress* the known loops by iterating through them as much as possible (given the symbolic constraints) and then observe the side effects in subsequent successor states. However, stressing every loop encountered in the trace is time consuming, so Bunkerbuster employs two strategies to prioritize loops that are more likely to lead to overflows.

First, not all loops write to memory and for the ones that do, not all writes rely on a changing pointer value or offset, which is necessary to cause an overflow. We coin this behavior as *stepping* and Bunkerbuster checks for instances of it in the recorded trace. Specifically, for each visit to each loop in the reconstructed CFG, Bunkerbuster collects the target memory address of each write instruction and examines how its target changes over each iteration. If there exists a write instruction such that each invocation targets an always increasing (or decreasing) memory address, then the loop is prioritized as a candidate for overflow analysis. An implementation of this algorithm is provided in the Appendix as Algorithm 2. Notice that since symbolic states are examined, the pointers can have multiple satisfiable values, so the satisfiability test for the stepping criteria is performed by the SMT solver.

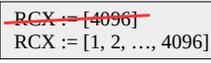
Next, the module takes into special consideration loops that engage in *counting* behavior because subsequent overflow candidates may have control dependencies to the computed value. For example, a string copying method can be implemented as two loops, the first counting how many bytes are in the string and the second copying them, as shown in Figure 5. If our algorithm blindly stresses the counting loop, its final written value will be maximized and then the subsequent copying loop will have to iterate the appropriate number of times. However, once the module detects that a code or return pointer in memory has been corrupted, continuing to stress the loop is excessive. Our solution is to detect counting loops, similarly to how stepping is detected, and replace the final value with a new symbolic variable constrained to all of the original’s intermediate values. For example, if the counting loop in Figure 5 can iterate up to 4,096 times, rather than constraining `length` to 4096, it is replaced with the symbolic integer set `[1, 4096]`. This allows it to discover the subsequent bug in fewer steps.

Once candidate loops have been stressed by iterating them as much as possible (or until a return pointer on the stack is overwritten), the module explores successor states until a return executes. If a control flow hijack is not detected by this point, it moves on to the next candidate until none remain.

Root Cause. To generate a report, the module first includes the basic block where the hijack occurred. Next, it identifies the memory

⁶See NetworkX’s `find_cycle` algorithm for a suitable implementation.

```

1. void my_strcpy(char *src, char *dst) {
2.     int length = 0;
3.     char *ptr = src;
4.     // "counting" loop
5.     while (*ptr) {
6.         ptr++;
7.         length++;
8.     }
9.     
10.    // "stepping loop"
11.    for (int i = 0; i < length; i++) {
12.        dst[i] = src[i];
13.    }
14.    dst[length] = 0;
15. }
16.
17. void foobar() {
18.     char *m_dst[128];
19.     char *m_src = {'A' * 4096, 0};
20.     my_strcpy(src, dst);
21. }

```

Figure 5: Counting loop example. Here the number of iterations of Line 12 depends on length, set by the loop starting at Line 5. When foobar passes my_strcpy a 4097 byte string, the register holding length (RCX) would normally become 4096 by Line 9. Our module overwrites RCX with a symbolic variable, allowing Line 11 to exit sooner, and then verifies the control hijack via a corrupted return pointer at Line 21.

location of the symbolic pointer that triggered the hijack using the symbolic constraints. An implementation of this algorithm is in the Appendix (Algorithm 3). Next, it rewinds backwards through the predecessor states until it finds the one that first made the pointer symbolic and adds it to the report. The module then generates a CDG for the execution path leading to this state, selects the nearest conditional branch in terms of shortest path, and checks the alternate branching states for contradicting constraints. If any are found, they become the preliminary patch for the developers, otherwise a new guarding branch should be placed before the corrupting state.

Format String Bugs. We find that unlike UAFs, DFs, and overflows, FS bugs are usually not as constrained by control flow. Specifically, if a call site contains a FS vulnerability, reaching it via *any* path is sufficient for discovering the bug. For this reason, we do not employ a tailored search strategy for FS and instead perform detection over the states found by the other exploration modules. In practice, format specifier strings should always be constant, turning them into read-only data at compile time. Consequently, for each call to a known format string function (e.g., printf), the module checks whether the specifier pointer or any of its content is symbolic. If it is, this means input data is able to directly control the specifier, which is a bug. In such cases, the root cause report identifies the caller of the format string function and the predecessor state that wrote to the specifier.

4 EVALUATION

We aim to answer the following questions in our evaluation:

- (1) *Is Bunkerbuster able to detect bugs within our covered classes?* We select 15 widely-used commodity programs and generate a corpus of benign inputs. After analysis, Bunkerbuster finds 39 bugs, of which 8 are new, never before reported cases. We manually verify the presence of all bugs. 1 EDB and 3 CVE IDs have been issued and patched by developers using Bunkerbuster's reports. We also measure Bunkerbuster's code coverage to show that its exploration converges.
- (2) *Is Bunkerbuster's exploration effective compared to prior techniques?* We compare against AFL [112] and QSYM [111] on our target programs, starting from similar seeds. After 1 week, Bunkerbuster finds 8 bugs missed by the other systems.
- (3) *Is Bunkerbuster's root cause analysis valuable compared to existing instrumentation?* We compare Bunkerbuster's root cause reports for Autotrace against those from QSYM with AddressSanitizer [91]. Bunkerbuster provides more accurate class labels in 4 cases.
- (4) *Are Bunkerbuster's exploration heuristics effective?* We compare the exploration techniques described in Section 3 against breadth-first and depth-first search and find that Bunkerbuster outperforms across all trials by better managing path explosion.
- (5) *Is Bunkerbuster 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, averaging 7.21% runtime overhead.
- (6) *Is Bunkerbuster's symbolic root cause analysis over partial paths correct?* We repeat the main experiment from the original symbolic root cause analysis work [106] using Bunkerbuster and verify our prototype produces the same results.

Experimental Setup. We use 1 computer to represent the end-host for tracing and 1 server to perform the analysis. Each device runs Debian Buster and contains an Intel Core i7-7740X processor, 32GB of memory, and solid state storage. Our prototype uses Angr [95] as its symbolic execution engine and is implemented in 7,062 Python and 1,208 C source lines of code (SLoC).

Dataset & Selection Criteria. To select our target programs for evaluation, we start by considering the packages offered in Debian's APT repository, filtered using the C/C++, CLI, and GUI tags, to ensure we only consider standalone programs written in languages that can contain memory corruption bugs. We then cross-reference MITRE's CVE database to isolate programs that contain or import (via libraries) code with known prior overflow, UAF, DF, and FS vulnerabilities, as these may contain more that have yet to be discovered. From this, we randomly pick 15 programs for testing.

We also manually assemble a corpus of benign inputs for each program by examining test cases and documentation. For CLI programs, we ensure the corpus has at least one case for each possible flag. For GUI programs, we manually perform some basic actions, such as opening, modifying, and saving files. When programs require complex input formats (e.g., images), we collect valid inputs from public sources like ImageNet [33].

Table 1: System Evaluation for Real-World Programs

ID	Type	Program	Component	Version	# Traces	# Novel	(%)	# Snaps	# BBs	# APIs	Find (s)
EDB-47254	Ovf	abc2mtex	main	1.6.1	1,209	166	13.7	166	124,248	26	27
CVE-2004-0597	Ovf	Butteraugli	libpng	1.2.5	176	78	44.3	78	1,648,790	112	15
CVE-2004-1257	Ovf	abc2mtex	main	1.6.1	1,209	166	13.7	166	50,120	26	97,188
CVE-2004-1279	Ovf	jpegtoavi	main	1.5	333	18	5.4	18	46,313	15	82,050
CVE-2013-2028	Ovf	Nginx	main	1.4.0	5	4	80.0	312	809,977	64	4,538
CVE-2009-5018	Ovf	gif2png	main	2.5.3	1,709	39	2.3	39	24,210,156	12	10
CVE-2017-7938	Ovf	dmitry	main	1.3a	10	10	100.0	10	56,488,245	20	78
CVE-2017-9167	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	23,764,196	84	2,659
CVE-2017-9168	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	74,252	84	3,868
CVE-2017-9169	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	74,753	84	728
CVE-2017-9170	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	74,543	84	2,868
CVE-2017-9171	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	33,965,824	84	786
CVE-2017-9172	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	95,561,159	84	6,038
CVE-2017-9173	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	33,965,824	84	5,995
CVE-2017-9191	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	23,070,692	84	5,364
CVE-2017-9192	Ovf	autotrace	libautotrace	0.31.1	55	37	67.3	37	23,764,196	84	3,010
CVE-2018-12326	Ovf	redis-cli	main	4.0.9	1,253	31	2.5	31	112,144	40	49
CVE-2018-12327	Ovf	ntpq	main	4.2.8p11	15	11	73.3	11	194,489	42	1,316
CVE-2018-18957	Ovf	GOOSE	libiec61850	1.3	5	2	40.0	6	65,198	9	11
CVE-2019-14267	Ovf	pdfressurrect	main	0.15	199	76	38.2	76	8,901,803	18	16,171
* CVE-2020-9549	Ovf	pdfressurrect	main	0.19	199	76	38.2	76	9,497,364	18	14,744
* CVE-2020-14931	Ovf	dmitry	main	1.3a	10	10	100.0	10	165,235	20	123
Will Not Fix	Ovf	GIMP	glibc	2.2.5	26	25	99.2	21,572	46,757,444	278	1
* CVE-2020-35457	Ovf	GIMP	glib	2.58.3	26	25	99.2	21,572	60,406,299	278	12
EDB-46807	Ovf	MiniFTP	main	1.0	7	3	42.8	33	60,849	45	7
* Patched	FS	dmitry	main	1.3a	10	10	100.0	10	125,662	20	16
CVE-2017-9162	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	95,561,159	84	3,253
CVE-2017-9163	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	90,334	84	3,253
CVE-2017-9182	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	30,386,474	84	2,873
CVE-2017-9183	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	413,022	84	3,253
CVE-2017-9190	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	40,806,309	84	3,253
CVE-2017-14103	UAF	GraphicsMagick	main	1.3.26	4	3	75.0	3	2,520,481	62	646
CVE-2019-17582	UAF	PHP	libzip	7.4.14	6	6	100.0	145	5,980,255	312	72
* Reported	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	40,632,944	84	3,253
* Reported	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	74,506	84	3,253
* Reported	UAF	autotrace	libautotrace	0.31.1	55	37	67.3	37	40,632,944	84	3,253
* EDB-49259	UAF	GIMP	babl	0.1.62	26	25	96.2	21,572	46,757,444	278	15
CVE-2017-11403	DF	GraphicsMagick	main	1.3.26	4	3	75.0	3	2,513,590	62	634
CVE-2017-12858	DF	PHP	libzip	7.4.14	6	6	100.0	145	5,980,255	312	72
Average:					189	36	64	1,710	19,392,602	90	7,045

* New vulnerability discovered by Bunkerbuster.

4.1 Bug Hunting in Real-World Programs

Methodology. For each of the 15 real-world programs in our dataset, we allow Bunkerbuster to trace and analyze our corpus of benign inputs for 1 week. We also measure Bunkerbuster’s code coverage over forwarded traces to test whether it converges, which is relevant to determining its usability in real-world deployments.

Results. Table 1 shows the results produced by Bunkerbuster’s analysis for the gathered data using our target programs and input corpus. In total, 39 bugs were found across the 15 tested programs. The “ID” column shows that 31 of the found bugs pertain to already publicly known vulnerabilities, whereas 8 have never been reported before. We manually inspect these cases to verify their presence. In 1 case, our prototype found a previously reported bug that the developers decided not to fix due to its performance consequences

versus the relatively low security impact. 1 bug has been issued an EDB ID by Offensive Security and 3 CVE IDs by MITRE. Developers have patched them, using our system’s reports to independently review and verify their novelty and impact. Some of these bugs were highly exploitable, including a now patched remote code execution (RCE) vulnerability, triggered via a WHOIS response.

The “Type” column lists the type of each bug. In total, Bunkerbuster found 25 overflows (Ovf), 1 FS bug, and 13 UAFs/DFs. The “Program” and “Component” columns report where the bugs reside, with “main” denoting the main executable object. 24 bugs were found within import libraries and 15 were inside the main object. We also report the version number of the vulnerable component for completeness. We observe that Autotrace is particularly buggy, with 17 vulnerabilities residing within the main object. Conversely, while GIMP is associated with 3 bugs, they were all found within

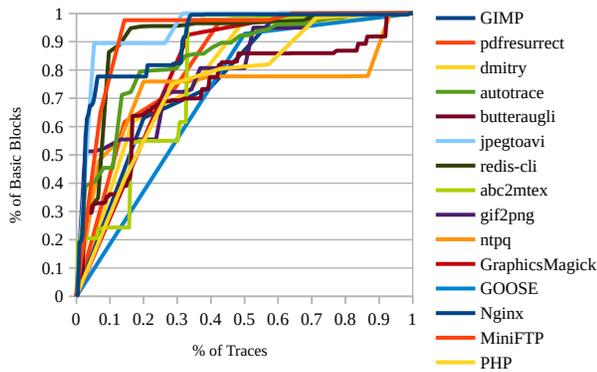


Figure 6: Basic block coverage for traces forwarded to the analysis, cumulatively.

imported libraries, demonstrating the importance of being able to analyze these APIs.

The “# APIs” column counts how many unique function imports were segmented by Bunkerbuster, using its symbolized memory snapshots and automatic prototype recovery. In other words, this is the number of APIs a human analyst would have to build scaffolding for if they were *not* using Bunkerbuster and wanted similar results. On average, 90 unique APIs were segmented per program, with counts ranging from 9 (GOOSE) to 278 (GIMP). Bunkerbuster eliminates the need to manually perform this laborious task.

The “# Traces” column reports how many traces (not segments) were recorded. On average, the end-host monitored 189 execution sessions per program. “# Novel” is the number of traces that contained at least 1 novel segment forwarded for analysis. On average, 36 where novel per program. For most programs, even with as few as 4 traces, at least 1 was filtered, demonstrating the importance of being able to identify and remove redundant data. The “# Snaps” column shows the number of trace segments and snapshots forwarded. On average, 1,710 were forwarded for analysis per program. In the case of GIMP, our input corpus yielded a comparatively high number of API snapshots. This is due to GIMP being one of the largest programs in our dataset, compiled from over 810,000 lines of C/C++ code, with a sophisticated architecture where each plugin is itself a standalone executable with additional library dependencies. For example, one of the `babl` functions found to contain a vulnerability was not invoked by GIMP directly, but rather by its plugin for loading PNG images. Trying to naively symbolically execute 46,757,444 basic blocks (from GIMP’s entry point, through the PNG plugin, into `babl`) would be difficult for prior work. Bunkerbuster succeeds thanks to its ability to segment.

“# BBs” records the number of traced basic blocks and “To Find” reports the number of seconds it took for the analysis to make its discovery. On average, traces containing bugs were 19,392,602 basic blocks long and bugs were found in 7,045 seconds, i.e., within 2 hours or so. Some bugs were found in as little as 1 second, while others took over 4 hours, depending on the complexity of the recorded behavior. Interestingly, because Bunkerbuster is able to segment and snapshot APIs, there is little correlation between trace length and the time to find bugs. For example, despite one GIMP trace

Table 2: Bunkerbuster Vs. AFL & QSYM

ID	Type	Program	BB	AFL	QSYM
EDB-47254	Ovf	abc2mtex	1	0	0
CVE-2004-1257	Ovf	abc2mtex	1	350	246
Patched	FS	dmitry	1	0	0
CVE-2020-14931	Ovf	dmitry	1	5	35
CVE-2020-9549	Ovf	pdfresurrect	1	0	0
CVE-2019-14267	Ovf	pdfresurrect	1	88	108
CVE-2017-11403	UAF	GraphicsM.	1	0	25
CVE-2017-14103	UAF	GraphicsM.	1	0	0
CVE-2018-12327	Ovf	ntpq	1	15	27
CVE-2018-12326	Ovf	redis-cli	1	18	44
CVE-2009-5018	Ovf	gif2png	1	88	163
CVE-2004-1279	Ovf	jpegtoavi	1	0	0
CVE-2004-0597	Ovf	Butteraugli	1	72	65
CVE-2018-18957	Ovf	GOOSE	1	1	1
CVE-2013-2028	Ovf	Nginx	1	0	0
EDB-46807	Ovf	MiniFTP	1	32	29
Will Not Fix	Ovf	GIMP	1	0	0
CVE-2020-35457	Ovf	GIMP	1	0	0
EDB-49259	UAF	GIMP	1	0	0
CVE-2019-17582	UAF	PHP	1	0	0
CVE-2017-12858	DF	PHP	1	7	7

being 60,406,299 basic blocks long, the bug it revealed was uncovered in 12 seconds. Conversely, several Autotrace traces of about 40,000,000 basic blocks each uncovered bugs in about 1 hour.

Figure 6 presents the coverage of our analysis over forwarded traces (i.e., after end-host-side filtering). To normalize each program’s curve, we present a cumulative distribution function (CDF) of the percentage of novel basic blocks discovered versus the percentage of traces analyzed. For all target programs, by the time 50% of the traces were analyzed, at least 80% of the total discovered basic blocks had been found, demonstrating that Bunkerbuster’s analysis converges. This is also consistent with the change in ratio of segments being filtered by the end-host over time.

4.2 Comparing Prior Exploration Techniques

Methodology. We compare Bunkerbuster against AFL [112], a highly popular greybox fuzzer, and QSYM [111], a recent concolic execution hybrid fuzzer, for this experiment. We pick these systems because they work in the binary-only setting for a wide range of bug classes, whereas other prior work requires source code [48] or is limited to a single class [59, 88], which would make for an unfair comparison. For consistency, we run each system on each target program for 1 week, starting from the same corpus of seeds. For each *unique* crash (as determined by AFL and QSYM), we manually inspect it to determine the bug class and root cause. We measure which bugs are detected by each system and how many reports are generated. We present the results for Autotrace in Subsection 4.3 as an extended case with crashes analyzed by AddressSanitizer.

Results. The results are presented in Table 2. In several cases, AFL and QSYM were unable to detect vulnerabilities found by Bunkerbuster. For example, they were unable to find the FS bug in DMitry because it requires a specific set of command line arguments to reliably cause a crash. Conversely, Bunkerbuster detected that

Table 3: Bunkerbuster Vs. AddressSanitizer

ID	Location	BB	QSYM + AS
CVE-2017-9167	input-bmp.c-337	Ovf	Ovf
CVE-2017-9168	input-bmp.c-353	Ovf	Ovf
CVE-2017-9169	input-bmp.c-355	Ovf	Ovf
CVE-2017-9170	input-bmp.c-370	Ovf	Ovf
CVE-2017-9171	input-bmp.c-492	Ovf	Ovf
CVE-2017-9172	input-bmp.c-496	Ovf	Ovf
CVE-2017-9173	input-bmp.c-497	Ovf	Ovf
CVE-2017-9191	input-tga.c-252	Ovf	Ovf
CVE-2017-9192	input-tga.c-528	Ovf	Ovf
CVE-2017-9162	autotrace.c-191	UAF	UNDEF
CVE-2017-9163	pxl-outline.c-106	UAF	UNDEF
CVE-2017-9182	color.c-16	UAF	UAF
CVE-2017-9183	autotrace.c-309	UAF	UNDEF
CVE-2017-9190	bitmap.c-24	UAF	BADFREE
Reported	pxl-outline.c-140	UAF	-
Reported	pxl-outline.c-609	UAF	-
Reported	color.c-10	UAF	-

symbolic format specifiers were being passed to `libc`, alerting it to the bug even in non-crashing cases. In general, we observed that the mutation algorithms used by AFL and QSYM are not well suited for fuzzing CLIs, which is also noted in AFL’s documentation. We also observe that of the 4 UAFs listed in Table 2, QSYM only found 1 and AFL none. QSYM and AFL also struggled to handle GIMP and GraphicsMagick due to their size and complexity, causing them to miss 10 and 11 bugs, respectively. It is possible that these tools would perform better if an expert human analyst created scaffolding around the imported libraries, but in GIMP’s case, there are *70 unique libraries* with *1,288 exported functions* to consider. Bunkerbuster relieves the analyst of this task.

In almost all of the cases where the prior systems found the same bug as Bunkerbuster, the former generated over 15 redundant reports. This is because AFL and QSYM rely on stack traces to determine the uniqueness of crashes, which are sometimes unreliable, such as when dealing with overflows. For example, QSYM generated *108 reports* for CVE-2019-14267 and *246* for CVE-2004-1257 because a stack corruption mislead it to classify each crash as unique. Bunkerbuster avoids this fatigue inducing redundancy using its symbolic root cause analysis, resulting in only 1 report per bug. Curiously, while QSYM generated more unique crashes than AFL overall, it only led to the discovery of 1 additional bug. This is likely due to the sparsity of bugs in real-world programs.

4.3 Comparing Prior Root Cause Techniques

Methodology. In this experiment, we perform the same evaluation as described in Subsection 4.2, with two adjustments made. First, we focus explicitly on Autotrace for this experiment because it yields by far the most bugs out of all the real-world programs. Second, we use AddressSanitizer (AS) to automatically triage the crashes uncovered by AFL and QSYM, as is common practice in real-world bug hunting. This allows us to compare the quality of Bunkerbuster’s root cause analysis to AS.

Over the course of this experiment, QSYM and AFL found 1 crash identified by AS as integer overflow and 1 out-of-bounds

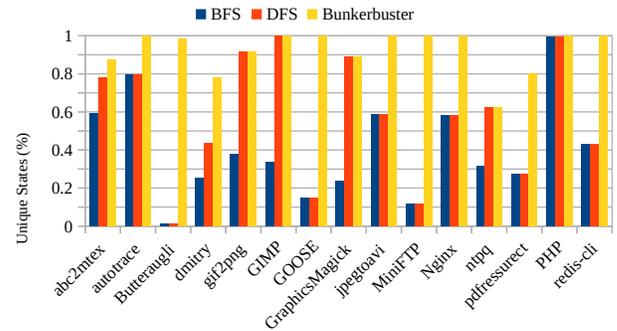


Figure 7: Percentage of unique basic blocks discovered using breadth-first search, depth-first search, and our proposed exploration techniques. Our techniques outperform the baselines across our entire dataset of 15 real-world programs.

read, which we exclude from the results since these are classes outside the current scope of Bunkerbuster. For clearer presentation, we translate binary addresses in our figures to source code line numbers using debug symbols, postmortem. *No system had access to the symbols during the experiment.* In our results, AFL and QSYM found the same set of bugs, so we only present QSYM for brevity.

Results. After 1 week of analysis, Bunkerbuster yields 17 bug findings. Conversely, QSYM yields 14 bugs after triaging by AS. Table 3 presents the two sets of reports side-by-side. Bunkerbuster finds all of the UAFs and overflows identified in the AS reports along with 3 UAFs never before reported. Upon investigation, we discover that the new UAFs reside in code branches missed by QSYM’s exploration. We believe that given more time, QSYM would eventually find inputs to reach these branches, whereupon AS would be able to triage them correctly. However, QSYM did not accomplish this within the allotted time whereas Bunkerbuster did.

Another interesting observation is that for 4 CVEs, Bunkerbuster is able to give more precise classifications than AS (bold in Table 3). In 3 cases, AS reports undefined behavior (UNDEF), meaning that despite QSYM detecting a crash and providing a concrete input to AS for analysis, AS still could not decide on a class for the bug. Conversely, Bunkerbuster correctly identifies the bugs to be UAFs. In 1 case, AS reports a bad free (BADFREE), meaning that the address being freed was never allocated, but Bunkerbuster, using its symbolic constraints, is able to correctly identify that a more carefully chosen input can turn this bug into a UAF. In summary, our system finds 3 UAFs missed by QSYM and yields more accurate classifications than AS in 4 cases.

4.4 Effectiveness of Exploration Techniques

Methodology. To validate whether our proposed exploration techniques enable Bunkerbuster to better search program states while avoiding path explosion, we compare against two baselines: breadth-first and depth-first search (BFS, DFS).⁷ Notice that DFS is the default exploration technique used by popular symbolic analysis frameworks [95].

⁷We include these baselines in the open sourced code repository for reproducibility.

To conduct the experiment, we randomly pick 1 trace for each of the real-world programs from our dataset and allow each technique (BFS, DFS, and ours) to explore states for 1 hour per program. Once the time limit has expired, we halt Bunkerbuster and count the number of unique basic blocks discovered by each technique. Since some target programs are slower to explore than others, we normalize our results by dividing the counts by the total number of unique blocks discovered globally, across all evaluated techniques, yielding a percentage from 0% to 100%.

Results. The results of our experiment are presented in Figure 7. Across all 15 real-world programs, Bunkerbuster’s exploration techniques outperform BFS and DFS. Specifically, for about half of the programs, Bunkerbuster’s techniques find all the basic blocks BFS and DFS find, and more. Bunkerbuster also finds more than double the number of basic blocks than the baselines in many cases, such as in Dmitry and MiniFTP.

The biggest contrast occurs in Butteraugli, where BFS and DFS only find about 2% of the blocks discovered by Bunkerbuster. Upon investigation, we discover that BFS and DFS both get stuck in libz’s CRC32 checksumming function. Such functions are notorious for inducing path explosion [25]. Bunkerbuster’s techniques avoid this function using heuristics to recognize that the contained code is unlikely to cause our targeted bug classes (e.g., the contained loops do not perform stepping writes, Subsection 3.5).

Another stark contrast occurs in MiniFTP, where the baselines only find about 10% of the blocks Bunkerbuster finds. In this case, BFS, DFS, and Bunkerbuster all focus on MiniFTP’s function for loading the settings file, which is expensive to explore because the code is densely packed with string comparisons, another well-known source of path explosion. However, whereas BFS and DFS explore this function naively, yielding lower code coverage and uncovering no bugs within the allotted time, Bunkerbuster prioritizes the contained loops using our described heuristics and finds EDB-46807 in under 10 seconds.

In summary, the heuristics we propose for Bunkerbuster do in fact help it explore more code in our evaluated dataset in less time than BFS or DFS. In many cases, the contrast is significant, with Bunkerbuster’s exploration techniques discovering more than double the number of basic blocks within the allotted time.

4.5 Performance & Storage

Methodology. To measure the performance and storage overheads of Bunkerbuster, we start with the SPEC CPU 2006 benchmark with a storage quota of 10 GB per end-host. We use the 2006 version deliberately so our numbers can be directly compared against other prior full-trace⁸ PT systems [40, 54]. Since these workloads are CPU intensive, we consider this to be the worst realistic case for our system. For another comparison point, we also evaluate Nginx running PHP with default settings, stressed using ApacheBench to serve 50,000 HTTP requests for files ranging in size from 100 KB to 100 MB, which we consider to be an I/O bound workload. Performance overhead is measured with tracing and API snapshots enabled versus running without the kernel driver installed for the baseline. Storage is the at-rest size of all collected data. Overheads

⁸As opposed to systems that use small finite buffers [31, 62].

Table 4: Symbolic Root Cause Verification

CVE / EDB	Type	# BBs	Δ RC	L	P	M
CVE-2004-0597	Ovf	41,625,163	247	Y	[3]	Y
CVE-2004-1257	Ovf	53,490	6,319	Y	-	-
CVE-2004-1279	Ovf	67,772	26,216	Y	-	-
CVE-2004-1288	Ovf	74,723	33,211	Y	[4]	Y
CVE-2009-2629	Ovf	300,071	28	Y	[7]	Y
CVE-2009-3896	Ovf	283,157	59	Y	[6]	Y
CVE-2009-5018	Ovf	90,738	1,848	Y	[42]	Y
CVE-2017-7938	Ovf	100,186	4,051	Y	-	-
CVE-2017-9167	Ovf	75,404	1,828	Y	-	-
CVE-2018-12326	Ovf	291,275	8	Y	[2]	Y
CVE-2018-12327	Ovf	374,830	122,740	Y	[78]	Y
CVE-2018-18957	Ovf	65,198	94	Y	[1]	Y
CVE-2019-14267	Ovf	128,427	83,123	Y	[80]	Y
EDB-15705	Ovf	260,986	19,322	Y	-	-
EDB-46807	Ovf	60,849	335	Y	-	-
CVE-2017-9182	UAF	132,302	296	Y	-	-
CVE-2017-11403	UAF	2,316,152	38	Y	[45]	Y
CVE-2017-14103	UAF	2,316,133	38	Y	[45]	Y
CVE-2017-12858	DF	5,980,255	51	Y	[69]	Y
CVE-2005-0105	FS	127,209	1	Y	[5]	Y
CVE-2012-0809	FS	108,442	1	Y	[100]	Y

are calculated as $(P - B)/B$ where B is the baseline metric and P is with Bunkerbuster.

Results. Figure 8 shows the metrics for the SPEC benchmark. The average tracing overhead is 7.21% with a geometric mean of 3.83%, which is within 1% of prior systems that record full PT traces, demonstrating that the filtering and snapshot steps performed by Bunkerbuster incur negligible additional overhead. Similar to prior work, the storage requirement is also large for some cases, averaging 1,348 MB/min, however all tests completed in under 1 minute, so the average final size is 110 MB per workload. We believe this is tolerable given that the data is forwarded to a storage server and with a 10 GB quota per end-host, dozens of executions can be stored at a time for analysis. Recall that this storage is temporary. Once a trace is analyzed, it can be discarded to free space. The bandwidth required to transfer traces currently makes Bunkerbuster better suited to enterprise LANs/WANs as opposed to end-hosts distributed across the internet.

Figure 9 shows the results for our Nginx benchmark. Here the average performance overhead is only 2% with 1.6 MB of data generated, on average, per HTTP request. With a quota of 10 GB, traces corresponding to thousands of requests can be buffered at a time. Requested file size had little impact on our results.

4.6 Verifying the Root Cause Analysis

Methodology. We trace proof of compromise exploits targeting overflow, UAF, DF, and FS vulnerabilities, for the same dataset used in the original symbolic root cause analysis work [106]. We then analyze the recorded traces with Bunkerbuster. In each case, we verified that our detection modules pinpoint the concise root cause of the vulnerability, in accordance with the prior work’s results.

Results. The results of our evaluation are summarized in Table 4, which shows the number of basic blocks in each trace, the number

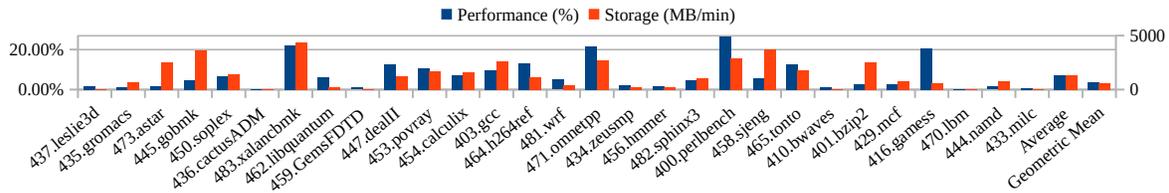


Figure 8: Performance and storage for tracing the SPEC CPU 2006 benchmark. The average overhead is 7.21% and the geometric mean is 3.83%. The average trace size is 1,348 MB/min and the geometric mean is 602 MB/min.

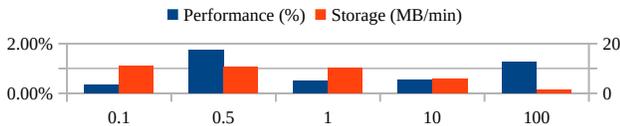


Figure 9: Overheads for tracing Nginx. The performance overhead is under 2% and the maximum storage is 1.6 MB per request.

of blocks between where the bug was detected and its determined root cause, whether the root cause was correctly located, whether a patch exists, and if so, whether the recommended constraints match the official patch. In total, 21 bugs were evaluated. As the table shows, Bunkerbuster’s detection modules are able to accurately detect and localize all 21 of the tested exploits, even when traces are over 1,000,000 basic blocks long and contain bugs that do not manifest into an observable corruption until over 100,000 blocks from the root cause. This gives us confidence that our symbolic root cause analysis is correctly designed, despite now working over partial traces in multi-path exploration.

5 LIMITATIONS & THREATS TO VALIDITY

Scope of Target Programs. Our current prototype is evaluated on benign, unobfuscated, Linux binaries. Further work is required to handle malware, packing, and virtualization, which fall outside the intended scope for this system. The current prototype also skips dynamically generated code (e.g., JIT compilation), however our driver is capable of recording and decoding it. Although our prototype focuses on Linux, the analysis is implemented for VEX IR, which is architecture independent and can be ported to other OSes that support PT, assuming the necessary system calls are modeled.

Scope of Detected Bug Classes. Bunkerbuster currently supports detection of overflow, UAF, DF, and FS bugs, but these are not the only types of memory corruption that can occur in programs written in unsafe languages like C/C++. However, all approaches to bug detection have class limitations. For example, the systems we compare against (AFL, QSYM) rely on crashes as indicators of buggy behavior, and consequently cannot detect non-crashing bugs, such as ones caught by exception handlers. Conversely, it is possible for Bunkerbuster to miss bugs that reside in program states that it cannot reach within the allotted time. It is also possible for Bunkerbuster to miss overflows that cannot corrupt the program counter. Detecting UAF, DF, and FS bugs relies on knowing which functions manage dynamic memory and accept format specifier strings in advance. The search strategies proposed in Section 3

are used only to prioritize certain paths and therefore do not limit Bunkerbuster’s total detection capabilities.

Reachability of Detected Bugs. As explained in Subsection 3.2, bugs found using snapshots taken from the program’s entry point are inherently reachable via input arguments. Conversely, bugs found via API snapshots may not be reachable via the analyzed program, but may be reachable by other programs that also import the same library. In such cases, we reported the bugs to the library maintainers, who decided to patch in most cases.

Severity of Detected Bugs. Our prototype does not currently analyze the exploitability of uncovered bugs, however our approach is compatible with automatic exploit generation techniques [11]. Our system has found confirmed 0-day RCE vulnerabilities, demonstrating the security relevance of our techniques.

In one case, Bunkerbuster found a bug that the developers decided not to patch, labeled “Will Not Fix” in Table 1. In this lone case, the developers acknowledged the bug’s existence, but decided that the performance cost of fixing it was too high, and instead cautioned downstream developers to take care in validating the inputs passed to the relevant library API.

6 PRIVACY & LEGAL CONSIDERATIONS

In the evaluation, we setup an end-host and an analysis server as separate machines to emphasize the decoupled nature of Bunkerbuster’s design. However, it is important to point out that analyzing control flow reveals some information about the values of data variables due to program control dependencies.

The threat of control flow leaking sensitive data has been well-studied by the side-channel research community [19], and some sensitive applications (e.g., cryptography) use hardened code to mitigate, however leakage in the context of traces recorded by PT has *not* been formally studied, to the best of our knowledge. Consequently, we envision the end-hosts and analysis servers belonging to the same or trusted parties where leakage is not an issue. However, it is possible for these machines to belong to different parties, raising privacy and legal concerns (e.g., Europe’s General Data Protection Regulation, a.k.a., GDPR). Further research is required to fully understand this risk, which is outside the scope of this work. Notice however that there exists prior work on sanitizing artifacts like crash dumps [35], some large corporations may already be recording PT traces from end-users [41], and once the analysis is distilled into a root cause report, its privacy risk diminishes, as can be seen in the example report shown in the Appendix (Figure 10).

7 RELATED WORK

Symbolic Analysis & Fuzzing. Early work in symbolic analysis proposed treating inputs as symbols to aid in testing code [18, 29, 64]. Over time, the applications of symbolic analysis expanded to include replaying protocols [75], vulnerability detection [44, 74, 88, 98], side-channel analysis [19], firmware analysis [94], verifying the correctness of cryptographic methods [24, 25], emulator testing [72], and automatic binary patching [77, 93]. Our work distinguishes itself from prior techniques like loop-extended symbolic execution (LESE) [88] in how it relies on novel uses of concrete data rather than a different type of symbolic lattice or grammar to achieve scalability. Whereas LESE has been evaluated on small CLI programs like Sendmail to uncover overflows, Bunkerbuster handles large plugin-based GUI tools like GIMP and also finds instances of orthogonal bug classes like UAF, DF, and FS. LESE cannot be extended to discover these classes because exploring loops is orthogonal to their life cycles.

An alternative approach to bug hunting is fuzzing [23, 36, 37, 43, 52, 56, 72, 85, 86, 110], which instead enumerates possible inputs to a program or API and checks for crashes as an indicator of buggy behavior. As mentioned in Section 1, some of the challenges with fuzzing are acquiring good seed inputs, reaching deep APIs, and identifying the nature of the bug when a crash does occur, typically using additional tools like AS. Although Bunkerbuster does not rely on fuzzing, it addresses the same usability challenges. While we consider the ability to collect traces from production systems with minor overhead to be a key novelty of our design, Bunkerbuster is technically capable of collecting traces from fuzzers as an alternative, should the user of our system not have access to production systems to monitor.

Many practical systems focus on *concolic execution*, whereby real executions are used to guide symbolic analysis without getting stuck in loops or string parsing [61, 90, 95]. Although Bunkerbuster also explores nearby paths with guidance from concrete data to discover vulnerabilities [22], our design takes a unique approach to avoiding path explosion. Namely, rather than turning to hybrid techniques that incorporate fuzzing [17, 27, 99, 111], source code [48], or prior crashes [82] to find more *inputs* (that can still lead to path explosion during symbolic analysis), we propose ways to leverage control flow *traces*. Bunkerbuster’s symbolic states enable it to detect a wide range of vulnerabilities (overflows, UAF, DF, FS) whereas prior taint-based tracing approaches are limited to a specific class, such as heap overflow [59]. Also, whereas many prior concolic systems have to operate in lockstep with the concrete environment [28, 59], Bunkerbuster’s tracing is completely decoupled from analysis, granting low overhead.

Automatic Harness Generation. In recent years, researchers have recognized the inability of fuzzers to handle large complex programs that are slow to initialize or require GUI interaction. Several proposals have emerged to automatically generate fuzzer harnesses using source code [12, 55]. Unfortunately, these solutions still leave COTS and legacy binaries unaddressed. In response, a system called Winnie [60] was proposed, which uses execution traces (rather than source code) to automatically generate harnesses for Windows binaries. At first glance, this appears comparable to how Bunkerbuster selectively symbolizes snapshots, however 5% of Winnie’s harnesses

had to be manually fixed to account for complex structures and callbacks while a large portion of the remaining 95% required minor manual tweaks. Bunkerbuster does not exhibit these shortcomings.

Root Cause Analysis. One of the oldest forms of root cause analysis is delta debugging [113], whereby comparisons are made between concrete program states for successful and failing inputs to pinpoint differences. Unfortunately, this requires having an ample number of test inputs in both classes to be effective. Alternatively, program slicing [87] can use tainting to identify the instructions that contribute to a failing instruction, even for a lone case, however the result can be hard to understand, with flagged instructions being sparsely scattered throughout the program. Conversely, Bunkerbuster’s root cause analysis leverages neighboring symbolic states, performing comparisons to pinpoint a concise root cause (unlike slicing), using symbolic constraints instead of concrete variables (unlike delta debugging), requiring only a single trace. Another approach to root cause analysis is failure sketching, however this is typically applied to bug classes like race conditions [63], or higher level issues in websites [8], insecure use of keys [68], and other domains outside Bunkerbuster’s scope [110].

It is also possible to produce root cause explanations by triaging the many crashes produced by tools like fuzzers into buckets of related cases. Bucketing can be done symbolically [81], semantically with program transformations [101], or statistically [16]. These lines of research are spiritual successors to delta debugging and carry similar limitations. Namely, they can only analyze bugs that result in a crash and require multiple crashing and non-crashing inputs to yield good explanations.

8 CONCLUSION

We propose Bunkerbuster, a system for automated data-driven bug hunting of memory corruption bugs using symbolic root cause analysis. Our design leverages PT and sparse memory snapshots to symbolically reconstruct execution traces and explore nearby paths to uncover overflow, UAF, DF, and FS vulnerabilities. We implement our prototype and evaluate it on 15 real-world Linux programs, where it finds 39 bugs, 8 of which are never before reported. 3 have been independently verified by MITRE, issued CVE IDs, and patched by developers using Bunkerbuster’s reports, validating our prototype’s usefulness. Bunkerbuster finds 8 bugs missed by AFL and QSYM in our target programs and correctly classifies 4 more bugs that AS mislabeled. Bunkerbuster achieves this with 7.21% performance overhead and reasonable storage requirements.

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] [n.d.]. Commit 074f7a8cd19b7661a59047e9257691df5470551c. <https://github.com/mz-automation/libiec61850/commit/074f7a8cd19b7661a59047e9257691df5470551c>. [Online; accessed 09-April-2020].
- [2] [n.d.]. Commit 9fdcc15962f9ff4baeb6fdd947816f43f730d50. <https://github.com/antirez/redis/commit/9fdcc15962f9ff4baeb6fdd947816f43f730d50>. [Online; accessed 16-January-2020].
- [3] [n.d.]. CVE-2004-0597 Patch. <https://github.com/mudongliang/LinuxFlaw/tree/master/CVE-2004-0597#patch>. [Online; accessed 25-October-2019].
- [4] [n.d.]. CVE-2004-1288 Patch. <https://pastebin.com/raw/fsFkspFF>. [Online; accessed 25-October-2019].
- [5] [n.d.]. CVE-2005-0105 Patch. <https://pastebin.com/raw/GHm1k1Rk>. [Online; accessed 25-October-2019].
- [6] [n.d.]. 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].
- [7] [n.d.]. Red Hat Bugzilla – Attachment 360889 Details for Bug 523105. <https://bugzilla.redhat.com/attachment.cgi?id=360889&action=diff>. [Online; accessed 07-January-2020].
- [8] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2016. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 641–652.
- [9] Jeff Arnold, Tim Abbott, Waseem Daher, Gregory Price, Nelson Elhage, Geoffrey Thomas, and Anders Kaseorg. 2009. Security impact ratings considered harmful. *arXiv preprint arXiv:0904.4058* (2009).
- [10] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful Rest API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 748–758.
- [11] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2018. Automatic Exploit Generation. Carnegie Mellon University.
- [12] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: Fuzz Driver Generation at Scale. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 975–985.
- [13] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 189–200.
- [14] Mihir Bellare and Bennet Yee. 1997. *Forward integrity for secure audit logs*. Technical Report. Computer Science and Engineering Department, University of California at San Diego.
- [15] Koustubha Bhat, Erik Van Der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2019. 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*. 545–558.
- [16] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *29th USENIX Security Symposium*. 235–252.
- [17] Konstantin Böttinger and Claudia Eckert. 2016. Deepfuzz: Triggering vulnerabilities deeply hidden in binaries. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 25–34.
- [18] Robert S Boyer, Bernard Elspas, and Karl N Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices* 10, 6 (1975), 234–245.
- [19] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. 2019. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In *CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation*. IEEE.
- [20] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. 2006. Towards automatic generation of vulnerability-based signatures. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE.
- [21] Frank Capobianco, Rahul George, Kaiming Huang, Trent Jaeger, Srikanth Krishnamurthy, Zhiyun Qian, Mathias Payer, and Paul Yu. 2019. Employing Attack Graphs for Intrusion Detection. In *New Security Paradigms Workshop (NSPW'19)*.
- [22] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 33rd Symposium on Security and Privacy (Oakland)*. San Francisco, CA.
- [23] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 725–741.
- [24] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. Sycmerts: Practical symbolic execution for exposing noncompliance in X.509 certificate validation implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 503–520.
- [25] Sze Yiu Chau, Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, and Ninghui Li. 2019. Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification. In *NDSS*.
- [26] Xi Chen, Asia Slowinska, and Herbert Bos. 2013. Who allocated my memory? Detecting custom memory allocators in C binaries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 22–31.
- [27] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. 2019. SAVIOR: Towards Bug-Driven Hybrid Testing. *arXiv preprint arXiv:1906.07327* (2019).
- [28] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices* 46, 3 (2011), 265–278.
- [29] Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering* 3 (1976), 215–222.
- [30] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. 2005. Vigilante: End-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles*. 133–147.
- [31] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA.
- [32] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios Kemerlis. 2016. Retracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. Austin, Texas.
- [33] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 248–255.
- [34] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.
- [35] Ren Ding, Hong Hu, Wen Xu, and Taesoo Kim. 2020. DESENSITIZATION: Privacy-Aware and Attack-Preserving Crash Report. In *Network and Distributed Systems Security (NDSS) Symposium 2020*.
- [36] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. 2014. KameleonFuzz: Evolutionary fuzzing for black-box XSS detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. 37–48.
- [37] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. 2020. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development (SecDev)*. IEEE, 23–30.
- [38] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. 1996. A sense of self for Unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*. 120–128. <https://doi.org/10.1109/SECPRI.1996.502675>
- [39] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*. Springer, 330–347.
- [40] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. 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.
- [41] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse Debugging of Kernel Failures in Deployed Systems. In *USENIX Annual Technical Conference*. 281–292.
- [42] gif2png. 2009. Command Line Buffer Overflow. <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=550978#50>. [Online; accessed 25-October-2019].
- [43] Patrice Godefroid. 2007. Random Testing for Security: Blackbox vs. Whitebox Fuzzing. In *Proceedings of the 2nd International Workshop on Random Testing*. 1–1.
- [44] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing. In *NDSS*, Vol. 8. Citeseer, 151–166.
- [45] GraphicsMagick. 2017. Attempt to Fix Issue 440. <http://hg.code.sf.net/p/graphicsmagick/code/rev/98721124e51f>. [Online; accessed 25-October-2019].
- [46] Philip J Guo and Dawson R Engler. 2009. Linux Kernel Developer Responses to Static Analysis Bug Reports. In *USENIX Annual Technical Conference*. 285–292.
- [47] David Habusha. [n.d.]. Vulnerability Prioritization Tops Security Pros' Challenges. <https://tinyurl.com/y6os685b>. [Online; accessed 18-November-2020].
- [48] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22nd USENIX Security Symposium*. 49–64.
- [49] Xueyan Han, Thomas Pasqueir, Adam Bates, James Mickens, and Margo Seltzer. 2020. Unicorn: Runtime Provenance-Based Detector for Advanced Persistent Threats. In *27th ISOC Network and Distributed System Security Symposium (NDSS'20)*.
- [50] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. 2019. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage. In *26th ISOC Network and Distributed System*

- Security Symposium (NDSS'19).*
- [51] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 380–394.
- [52] Renáta Hodován and Ákos Kiss. 2016. Fuzzing javascript engine apis. In *International Conference on Integrated Formal Methods*. Springer, 425–438.
- [53] Jason E. Holt. 2006. Logcrypt: Forward Security and Public Verification for Secure Audit Logs. In *Proceedings of the Australasian Information Security Workshop (AISW-NetSec)*.
- [54] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William Harris, Taesoo Kim, and Wenke Lee. 2018. 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.
- [55] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. Fuzzgen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium*. 2271–2287.
- [56] Joonun Jang and Huy Kang Kim. 2019. FuzzBuilder: Automated building grey-box fuzzing environment for C/C++ library. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 627–637.
- [57] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. 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.
- [58] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. 2017. Towards Efficient Heap Overflow Discovery. In *26th USENIX Security Symposium*. 989–1006.
- [59] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. 2017. Towards Efficient Heap Overflow Discovery. In *26th USENIX Security Symposium*. 989–1006.
- [60] Junho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Network and Distributed System Security Symposium*.
- [61] Min Gyung Kang, Stephen McCamant, Pongsin Pooankam, and Dawn Song. 2011. Dta++: dynamic taint analysis with targeted control-flow propagation.. In *NDSS*.
- [62] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 582–598.
- [63] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. 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.
- [64] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
- [65] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. 2005. Enriching Intrusion Alerts Through Multi-Host Causality.. In *Proceedings of the 12th ISOC Network and Distributed System Security Symposium (NDSS'05)*.
- [66] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*. 1–6.
- [67] Yonghwi Kwon, Brendan Saltaformaggio, I Luk Kim, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. A2C: Self Destructing Exploit Executions via Input Perturbation. In *Network and Distributed Systems Security (NDSS) Symposium 2017*.
- [68] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. 2018. K-Hunt: Pinpointing insecure cryptographic keys from execution traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 412–425.
- [69] libzip. 2017. Fix double free. <https://github.com/nih-at/libzip/commit/2217022b7d1142738656d891e00b3d2d9179b796>. [Online; accessed 25-October-2019].
- [70] Philip Low. 2017. Insuring against cyber-attacks. *Computer Fraud & Security* 2017, 4 (2017), 18–20.
- [71] Di Ma and Gene Tsudik. 2009. A new approach to secure logging. *ACM Transactions on Storage (TOS)* 5, 1 (2009).
- [72] Lorenzo Martignoni, Stephen McCamant, Pongsin Pooankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *ACM SIGARCH Computer Architecture News*, Vol. 40. ACM, 337–348.
- [73] S. Momeni Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrisnan. 2019. HOLMES: Real-Time APT Detection through Correlation of Suspicious Information Flows. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, USA.
- [74] David A Molnar and David Wagner. 2007. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. *UC Berkeley EECS* (2007).
- [75] James Newsome, David Brumley, Jason Franklin, and Dawn Song. 2006. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 311–321.
- [76] James Newsome, David Brumley, Dawn Song, Jad Chamcham, and Xeno Kohvah. 2006. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software.. In *NDSS*.
- [77] Anh Nguyen-Tuong, David Melski, Jack W Davidson, Michele Co, William Hawkins, Jason D Hiser, Derek Morris, Ducson Nguyen, and Eric Rizzi. 2018. Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge. *IEEE Security & Privacy* 16, 2 (2018), 42–51.
- [78] ntp. 2018. 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].
- [79] Riccardo Paccagnella, Kevin Liao, Dave (Jing) Tian, and Adam Bates. 2020. 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 (CCS'20)*.
- [80] pdfresurrect. 2019. Prevent a buffer overflow in possibly corrupt PDFs. <https://github.com/enferex/pdfresurrect/commit/4ea7af4f51d0440da651d099247e2273f811dbc>. [Online; accessed 25-October-2019].
- [81] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. 2017. Bucketing Failing Tests via Symbolic Analysis. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 43–59.
- [82] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. 2015. Hercules: Reproducing Crashes in Real-World Application Binaries. In *37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 891–901.
- [83] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security 19)*. 1733–1750.
- [84] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*. 869–886.
- [85] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.
- [86] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*. 861–875.
- [87] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. 2013. Using likely invariants for automated software fault localization. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 139–152.
- [88] Prateek Saxena, Pongsin Pooankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 225–236.
- [89] Bruce Schneier and John Kelsey. 1998. Cryptographic Support for Secure Logs on Untrusted Machines.. In *Proceedings of the USENIX Security Symposium (USENIX)*.
- [90] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 263–272.
- [91] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC)*. 309–318.
- [92] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 329–339.
- [93] Yan Shoshitaishvili, Antonio Bianchi, Kevin Borgolte, Amat Cama, Jacopo Corbetta, Francesco Disperati, Audrey Dutcher, John Grosen, Paul Grosen, Aravind Machiry, et al. 2018. Mechanical phish: Resilient autonomous hacking. *IEEE Security & Privacy* 16, 2 (2018), 12–22.
- [94] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.. In *NDSS*.
- [95] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. 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.
- [96] Xiaokui Shu, Danfeng (Daphne) Yao, Naren Ramakrishnan, and Trent Jaeger. 2017. Long-Span Program Behavior Modeling and Attack Detection. *ACM Transactions on Privacy and Security* 20, Article Article 12 (2017).
- [97] Asia Slowinska and Herbert Bos. 2007. 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, 487–500.
- [98] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Pooankam, and Prateek Saxena.

2008. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*. Springer, 1–25.

[99] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[100] sudo. 2012. Format String Vulnerability. <https://bugs.gentoo.org/401533>. [Online; accessed 25-October-2019].

[101] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic Crash Bucketing. In *33rd IEEE International Conference on Automated Software Engineering*. IEEE, 612–622.

[102] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Jungwhan Rhee, Zhengzhang Zhen, Wei Cheng, Carl A. Gunter, and Haifeng chen. 2020. You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis. In *27th ISOC Network and Distributed System Security Symposium (NDSS'20)*.

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

[104] Andreas Wespi, Marc Dacier, and Hervé Debar. 2000. Intrusion Detection Using Variable-Length Audit Trail Patterns. In *Recent Advances in Intrusion Detection*. Springer, 110–129.

[105] K. Xu, K. Tian, D. Yao, and B. G. Ryder. 2016. 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)*. 467–478.

[106] Carter Yagemann, Matthew Pruett, Simon P. Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *USENIX Security Symposium*.

[107] Carter Yagemann, Salmin Sultana, Li Chen, and Wenke Lee. 2019. Barnum: Detecting Document Malware via Control Flow Anomalies in Hardware Traces. In *Proceedings of the 25th Information Security Conference (ISC)*. New York, NY, USA.

[108] Attila Altay Yavuz and Peng Ning. 2009. BAF: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

[109] Attila A Yavuz, Peng Ning, and Michael K Reiter. 2012. 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)*.

[110] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2139–2154.

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

[112] Michal Zalewski. 2017. American Fuzzy Lop. <http://lcamtuf.coredump.cx/afl> (2017).

[113] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

Figure 10: Example root cause report for CVE-2018-12326.

```
Trace: openhost+0x2a4 in ntpq (0xbae4)
Trace: openhost+0x218 in ntpq (0xba58)
Trace: openhost+0x3bc in ntpq (0xbbfc)
Trace: __stack_chk_fail+0x0
We've triggered a bug
Analyzing exit at openhost+0x218
Blaming: openhost+0x2dd in ntpq (0xbb1d)
Recommendation: Add [argv[232] == ']'] to
<CFGENode openhost+0x2d8 0x55857a27db18[5]>
Vulnerability Hooks Details:
Hash: 1cfad
Addr: 0x55857b000028 => __stack_chk_fail+0x0
      0x55857a27dc01 => openhost+0x3c1
```

Table 5: Manually Verified APIs for Binary-Only Recovery

Library	# Functions	# Variables	# Pointers	Match?
libpng	71	183	117	Yes
libz	11	14	2	Yes
glib	125	283	202	Yes
libc	22	29	19	Yes
libbabl	70	163	104	Yes
libx11	5	247	137	Yes
libjpeg-turbo	25	15	12	Yes
libcyrus-sasl	1	3	1	Yes
libpoppler	3	7	4	Yes
libgegl	33	52	44	Yes
libghostpdl	29	47	40	Yes
libgimp	36	50	48	Yes
libgtk	21	41	26	Yes
libkeyutils	4	8	6	Yes
libidn2	1	2	1	Yes
libXpm	5	18	5	Yes
libopenjpeg	22	40	24	Yes
Total:	484	1,202	792	

Algorithm 1: Retrieve all memory reads and writes for a VEX IRSB I , using successor state S , producing A .

```

1  $A, T \leftarrow \emptyset$ 
2 foreach  $i \in I$  do
3   if  $\text{Type}(i) = \text{Store}$  then
4     if  $\text{Type}(i.\text{addr}) = \text{Const}$  then
5       // Write to constant address
6        $A \leftarrow A \cup i.\text{addr}$ 
7     end
8     else
9       // Write to variable address
10       $T \leftarrow T \cup i.\text{addr}$ 
11    end
12  if  $\text{Type}(i) = \text{WrTmp} \wedge \text{Type}(i.\text{data}) = \text{Load}$  then
13    if  $\text{Type}(i.\text{data}.\text{addr}) = \text{Const}$  then
14      // Read from constant address
15       $A \leftarrow A \cup i.\text{data}.\text{addr}$ 
16    end
17    else
18      // Read from variable address
19       $T \leftarrow T \cup i.\text{data}.\text{addr}$ 
20    end
21  end
22 end
23 // Use  $S$  to avoid recomputing ASTs
24 foreach  $t \in T$  do
25    $A \leftarrow A \cup \text{EvalTmp}(S, t)$ 
26 end

```

Algorithm 2: Detect stepping behavior in a sequence of states S , iterating a loop. IsTmpStore is true when the VEX IRSB instruction is a WrTmp and its expression is Store .

```

1  $R \leftarrow \text{False}$ 
2  $I \leftarrow \emptyset$ 
3 foreach  $s \in S$  do
4   foreach  $i \in s.\text{irsb}.\text{statements}$  do
5     if  $\text{IsTmpStore}(i)$  then
6        $I[i.\text{addr}] \leftarrow I[i.\text{addr}] \cup i.\text{addr}.\text{tmp}$ 
7     end
8   end
9 end
10 foreach  $a \in I$  do
11    $l \leftarrow I[a].\text{size}$ 
12   if  $l > 1$  then
13     if  $I[a][0] \leq I[a][1] \leq \dots \leq I[a][l]$  then
14        $R \leftarrow \text{True}$ 
15     end
16     if  $I[a][0] \geq I[a][1] \geq \dots \geq I[a][l]$  then
17        $R \leftarrow \text{True}$ 
18     end
19   end
20 end

```

Algorithm 3: Tainting algorithm to obtain the registers and addresses used to calculate a VEX IR temporary variable.

```

1 Input: VEX IR statements  $S$  starting from last executed.
2 Tmp  $n$  to taint initially.
3 Result: Addresses  $A$  and registers  $R$  used to calculate  $n$ .
4  $A, R \leftarrow \emptyset$ 
5  $T \leftarrow \{n\}$ 
6 foreach  $s$  in  $S$  do
7   if  $\text{Type}(s) = \text{Put}$  and  $\text{Type}(s.\text{data}) = \text{RdTmp}$  then
8     if  $s.\text{data}.\text{tmp} \in T$  then
9        $R \leftarrow R \cup \{s.\text{register}\}$ 
10    end
11  end
12  if  $\text{Type}(s) = \text{WrTmp}$  and  $s.\text{tmp} \in T$  then
13    foreach  $a$  in  $s.\text{data}.\text{args}$  do
14      if  $\text{Type}(a) = \text{Get}$  then
15         $R \leftarrow R \cup \{a.\text{register}\}$ 
16      end
17      if  $\text{Type}(a) = \text{RdTmp}$  then
18         $T \leftarrow T \cup \{a.\text{tmp}\}$ 
19      end
20      if  $\text{Type}(a) = \text{Load}$  then
21         $A \leftarrow A \cup \text{EvalTmp}(a.\text{address})$ 
22      end
23    end
24  end

```
