A Flexible Framework for Expediting Bug Finding by Leveraging Past (Mis-)Behavior to Discover New Bugs

Sanjeev Das*
IBM Research
sanjeev.das@ibm.com

Kedrian James
UNC Chapel Hill
kedjames@cs.unc.edu

Jan Werner
UNC Chapel Hill
jjwerner@cs.unc.edu

Manos Antonakakis
Georgia Tech
manos@gatech.edu

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

Fabian Monrose
UNC Chapel Hill
fabian@cs.unc.edu

ABSTRACT

Among various fuzzing approaches, *coverage-guided* grey-box fuzzing is perhaps the most prominent, due to its ease of use and effectiveness. Using this approach, the selection of inputs focuses on maximizing program coverage, *e.g.,* in terms of the different branches that have been traversed. In this work, we begin with the observation that selecting *any* input that explores a new path, and giving *equal* weight to all paths, can lead to severe inefficiencies. For instance, although seemingly "new" crashes involving previously unexplored paths may be discovered, these often have the same root cause and actually correspond to the same bug.

To address these inefficiencies, we introduce a framework that incorporates a tighter feedback loop to guide the fuzzing process in exploring truly diverse code paths. Our framework employs (*i*) a vulnerability-aware selection of coverage metrics for enhancing the effectiveness of code exploration, (*ii*) crash deduplication information for early feedback, and (*iii*) a configurable input culling strategy that interleaves multiple strategies to achieve comprehensiveness. A novel aspect of our work is the use of hardware performance counters to derive coverage metrics. We present an approach for assessing and selecting the hardware events that can be used as a meaningful coverage metric for a target program. The results of our empirical evaluation using real-world programs demonstrate the effectiveness of our approach: in some cases, we explore fewer than 50% of the paths compared to a base fuzzer (AFL, MOpt, and Fairfuzz), yet on average, we improve new bug discovery by 31%, and find the same bugs (as the base) 3.3 times faster. Moreover, although we specifically chose applications that have been subject to recent fuzzing campaigns, we still discovered 9 new vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Vulnerability management**.

*The research was conducted while the author was a postdoc at UNC Chapel Hill.

KEYWORDS

Fuzzing, Machine Learning, Hardware Performance Counters

ACM Reference Format:
Sanjeev Das, Kedrian James, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2020. A Flexible Framework for Expediting Bug Finding by Leveraging Past (Mis-)Behavior to Discover New Bugs. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3427228.3427269

1 INTRODUCTION

In recent years, fuzz testing (or fuzzing) [30] has emerged as the preeminent automated technique for finding vulnerabilities in software. Generally speaking, the process of fuzzing involves feeding crafted input to a program in the hope of triggering unhandled exceptions and crashes. Today, so-called greybox fuzzing has been very effective in finding vulnerabilities in real-world programs.

The success of greybox fuzzers like the American Fuzzy Lop (AFL) [61] stems from the fact that they use a feedback loop to prioritize the inputs fed to a program. The overall process involves input selection, scheduling, and mutation. In the first stage, inputs are tested, and based on various feedback mechanisms, interesting inputs (*i.e.,* those that crashed the program or led to new paths being explored) are chosen for mutation. The mutation stage typically assumes input data as a sequence of bytes, and performs operations such as bit or byte flipping, increment/decrement of integer data, and so on. Input selection and scheduling have been shown to be critically important [44, 57] because they govern the doctrine that fuzzers apply in their search for vulnerabilities [31, 32]. As it pertains to the input selection process, contemporary greybox fuzzers use a coverage-driven principle. *Coverage-guided fuzzing (CGF)* approaches select inputs that increase the total program coverage. For example, AFL uses branch coverage to steer input selection. As such, it only selects those inputs that explore a new branch that was not traversed before. The inherent goal of a CGF approach is to try to increase the code coverage in order to eventually reach a path that may stumble upon a vulnerability in the program. Indeed, the success of AFL [61], and extensions thereof [4, 48, 50], is largely attributable to the use of code coverage as feedback [10].

Unfortunately, blindly adopting such a strategy can be inefficient because: (*i*) *any* input that covers a new path is selected, and (*ii*) *equal* weight is given to each path. To see why this matters, consider Figure 1 which depicts an offline analysis of crashes generated by AFL. The topmost line reports the "uniqueness" of the generated crashes based on the branch edges explored. The fuzzer generates a lot of crashes, but not all crashes are created equal [23]. Indeed, a straightforward grouping of crashes using a stack trace approach shows that the number of unique crashes is significantly lower than AFL indicates. More importantly, these crashes are obtained intermittently (as depicted by the red line), likely due to time wasted exploring closely related paths or getting stuck in deep code paths. The fruitless search wastes time on inefficient mutation operations that do not lead to different code paths, ultimately failing to finding so-called "quality inputs" that lead to crashes for a long time [27]. Furthermore, Böhme et al. [2] showed that although finding some vulnerabilities using contemporary approaches might be cheap, improving bug finding *linearly* requires *exponentially* more computational power.

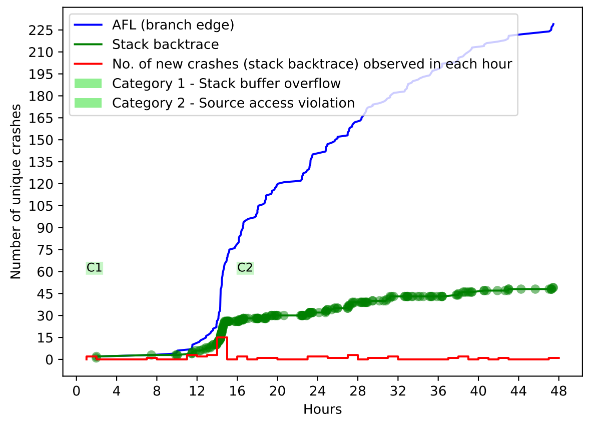


Figure 1: Inefficiencies in the typical fuzzing process

Additionally, since most of the program paths are (hopefully) bug-free [33, 51], giving equal value to all program paths is a non-optimal strategy when the ultimate goal is to discover vulnerabilities [9]. This issue is clearly visible in Figure 1, which also shows the time when the first instance of each of only two classes of bugs (a stack overflow and a source access violation vulnerability, respectively) were found. With Chen et al. [9]'s observation about path coverage in mind, other researchers have proposed ways for improving code coverage by using different coverage metrics [54] (*e.g.*, context-sensitive branch coverage [7, 43], memory-access-aware branch coverage [16], basic block information [25, 43]) have been proposed. Unfortunately, Wang et al. [54] subsequently demonstrated that there is *no grand slam coverage metric* that outperforms all the others. That is, given limited resources (time and computation power), all of these coverage metrics offered merit relative to flipping certain types of branches or in finding vulnerabilities not readily found by the others. Alas, different coverage metrics performed better on other benchmarks, underscoring the enormous diversity that exists between the code base of different programs. Thus, even though a *one size fits all* coverage metric may be desirable, doing so is unlikely to discover diverse bug types across a wide variety of programs.

Taken together, these findings open the door for several areas of improvement within the practice of fuzzing. For one, by introducing a tighter feedback loop early on in the workflow, one can better guide a fuzzer in exploring more diverse code paths. Likewise, armed with the ability to dynamically choose *when* to apply a particular coverage metric, we may be able to improve the overall success rate of a fuzzer by steering it toward or away from certain classes of bugs.

Our specific contributions include:

1. *Vulnerability-aware selection of a coverage metric:* we use micro-architectural information obtained through hardware performance counters (HPCs) to derive coverage metrics. HPCs are a set of special registers that are available in processors to monitor and measure hardware events related to memory, branches, instructions, and basic blocks. We provide a principled approach for systematically assessing hundreds of HPC events to select representative sets of events that can be utilized as a coverage metric for a given program.
2. *A configurable input selection strategy:* using knowledge of past bugs, we show how one can perform fuzzing under two modes of operation — to seek bugs that are induced under similar behavior as witnessed in past bugs, or to hunt for bugs that are triggered by program behavior that is markedly *different* from that observed when previous vulnerabilities were discovered. Given the large number of events that can be utilized to build a coverage metric, there is tremendous flexibility in exploring input selection strategies at runtime, *e.g.*, by choosing a different set of HPC events than those that performed best in teasing out past bugs. We leverage this flexibility to switch between coverage metrics at runtime (*e.g.,* when the deduplication strategy informs us that the last few crashes likely fall under the same bug classification).
3. *Using deduplication as a feedback mechanism:* we show that deduplication can be used as an early feedback mechanism to improve the overall fuzzing progress. Specifically, using crash deduplication techniques to quickly infer the potential root cause and to steer the fuzzing process, one can lessen the chances of discovering the same bugs repeatedly.
4. *Extensibility:* We demonstrate that our approach can help improve the effectiveness of different base fuzzers. Moreover, we perform an extensive evaluation based on practitioner's vantage point, in terms of time to finding a *unique* crash, and the consistency of bug discovery over repeated runs. To demonstrate the benefits of our extensions to the vulnerability discovery process, we report on an evaluation on eight real-world programs that were specifically chosen because they have been subject to heavy fuzzing attempts in the recent past.

2 BACKGROUND

In what follows, we provide an overview of hardware performance counters (HPC), as the usage of these counters plays a key role in our approach. In short, hardware performance counters are a set of special registers present in modern processors that can be used to monitor and measure events at the hardware level. These events are related to instructions, memory, and the execution behavior on the

CPU pipeline. The hardware events supported by performance counters can be classified as either architectural or non-architectural events (the latter are also known as micro-architectural events).

Architectural events comprise events that remain consistent across different processor architectures. Examples include instructions, branches, and cycles. Non-architectural events are those that are specific to the micro-architecture of a given processor, for example, cache accesses, branch prediction, and TLB accesses. Unlike architectural events, non-architectural events vary among processor architectures and may also change with processor enhancements. Table 8 in the §A.1 presents a list of commonly used architectural and non-architectural events in Intel processors. Interested readers are referred to [14, 34, 46, 52] for excellent overviews of performance counters and their proper usage.

In this work, we argue that the wealth of information provided by performance counters about program behavior offers a unique opportunity to explore a multitude of coverage metrics to guide the fuzzing progress. Moreover, since these counters can be used to monitor more than one event simultaneously, one can build custom coverage metrics to suit different proposes. From a practitioner's point of view, the collection of these characteristics comes with low overhead, making HPCs extremely well suited for the task at hand.

Intuitively, our entire approach hinges on the assumption that bugs in prior versions of a program or library can provide a good enough signal to help find bugs in other versions of the same program [37]. Indeed, the seminal work of Ozment and Schechter showed that much of the "foundational code" remains the same in the newer versions of a program [40]. Moreover, because similar coding practices are followed in newer versions of a program, coding mistakes tend to persist. Based on that observation, we provide a framework that can operate in two ways: steer the fuzzing process toward or away from paths that have architectural events similar to those observed when prior bugs were discovered. To allow for this flexibility, we provide a way to systematically assess the HPC events and select representative sets of events (up to four at a time) that can be used to differentiate between quality (*i.e.,* crashing) and non-quality inputs. Next, models are built using standard machine learning approaches. Lastly, at runtime, we apply these models to help drive the input selection strategy while fuzzing other versions (albeit past or current) of the program that were not part of the learning phase.

3 OUR APPROACH: OMNIFUZZ

In this work, we propose a flexible vulnerability-driven fuzzing framework, called *OmniFuzz*, for enhancing baseline fuzzers in order to find bugs faster, and hopefully, to find more unique bugs. Unlike contemporary approaches that give equal weight to all paths explored, we prioritize code paths based on knowledge of past bugs. By judiciously selecting what inputs should be selected for mutation, we streamline the space of paths that need to be explored, thereby minimizing the amount of time spent on paths that are unlikely to lead to successful outcomes in our quest to locate bugs.

OmniFuzz operates in three phases (illustrated in Figure 2). In the data collection phase, we collect quality and non-quality inputs. In fuzzing parlance, quality inputs are those that trigger crashes, whereas non-quality inputs exit the program normally and do not result in crashes or hangs. Next, we extract the dynamic behavior for all the inputs using hardware performance counters. In the model building phase, we use the HPC traces to select an appropriate coverage metric for a given program. The metric and the HPC traces are used to build a machine learning (ML) based classifier, which is trained to identify the quality and non-quality inputs. During the runtime phase, the classifier guides the fuzzing of a new version of the program. We discuss each phase in turn.

3.1 Data Collection

To collect quality inputs, we subject a version of the program to a baseline fuzzer. The number of quality inputs generated by a fuzzer depends on multiple factors, including the number of actual bugs, the seed inputs, and the amount of available computing resources. To maximize the number of quality inputs, we run the baseline fuzzer with sufficient computing resources over an extended period [17, 28]. Since modern fuzzers typically generate a large number of test inputs quickly, collecting a corpus of non-quality inputs is not an issue. Next, we curate HPC traces for both the quality and non-quality inputs. The traces consist of the number of occurrences of a specific hardware event that was triggered during the program execution.

Although there are hundreds of hardware events available in modern processors, only a limited number of counters can be used to monitor these events simultaneously. Given this constraint, a natural question is which events should one use? To answer that question, we conducted an in-depth analysis of processor documentation, and so studied the literature for events that were commonly used to profile program behavior [5, 29, 38, 56]. We conservatively selected all those events that could explain the high-level behavior of a program, but excluded those events that monitor low-level micro-architectural information or that are difficult to relate to high-level program behavior. For example, we excluded the events relating to the pipelining behavior of the CPU. In the end, we were left with a set of 96 events, which we further grouped into 65 classes (shown in Table 8). The criteria we used to group events were (*a*) events that are similar but only differ due to the change in the size of hardware components are considered as a single class, and (*b*) events that count hits instead of cycles are split into different classes. To collect the HPC data, we followed the recommendations of Das et al. [14].

3.2 Model Building

We use a correlation-based technique to identify representative sets (of 4 events) from the initial set of 96 events. Here, a good feature set contains features that are correlated with a class, yet uncorrelated with each other. In our context, a class represents either quality or non-quality inputs. Given good feature sets, we now need a way to influence the input selection at runtime. Based on prior knowledge of the non/quality inputs, one might be able to predict whether some unknown input is a quality or non-quality input by learning the dynamic program behavior for these inputs. However, in the context of fuzzing, the problem is a bit more complicated because we need to predict whether fuzzing with the *current input might lead to a crash in some future* iteration.

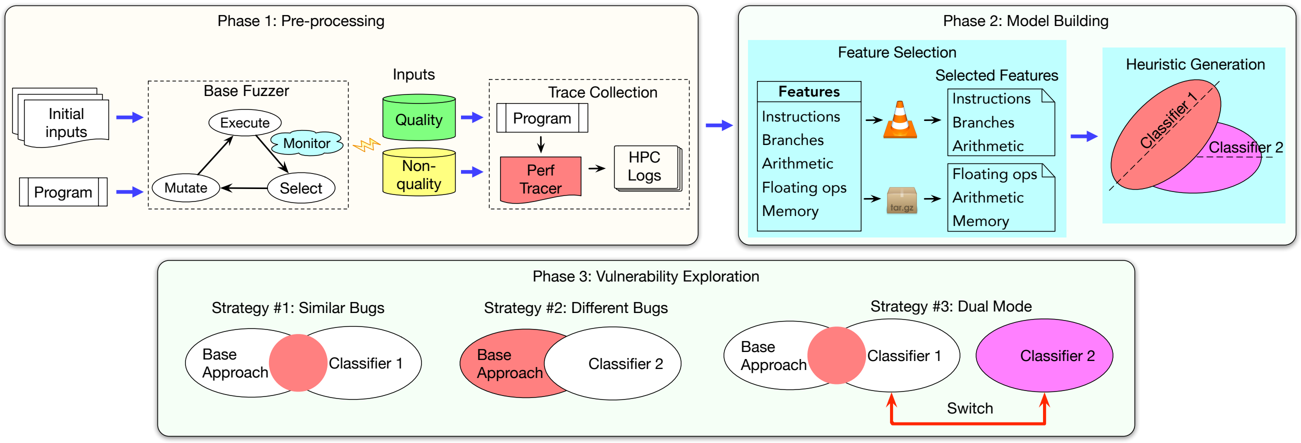


Figure 2: System design of OmniFuzz

To address this challenge, we use a machine learning approach that learns the dynamic program behavior triggered by the quality inputs, and guides the fuzzing process accordingly. As part of the exploratory phase of our research, we performed model selection using several machine learning algorithms, including decision trees, random forests, k-nearest neighbors, and multi-layer perceptrons (MLPs). We found that no particular model significantly outperformed the rest. However, we settled on using the MLP approach for a number of reasons. First, it does not depend on the discrete numerical value of features (in contrast to decision trees). Second, it has constant runtime overhead because our neural network model is small; specifically, it contains a fixed number of nodes (typically 4 neurons in the input layer, 4 neurons in the hidden layer and 2 neurons in the output layer).

3.3 On-the-fly Deduplication

To limit time wasted hitting the same or similar bug(s) repeatedly, we use root cause analysis as a deduplication mechanism. The root cause, in our case, is the line or block of code that first propagated the bad value that led to the crash. To support online deduplication via root cause analysis, we implemented a custom solution on top of Mozilla's Record and Replay framework [35]. The framework provides a wealth of information that enables precise analysis of the program states that led to a crash. We also leverage the use of hardware watchpoints for memory tracking. Our memory tracking technique is based on the pointer lifecycle in Figure 3. To locate the root cause of a crash, analysis begins at the crash point by extracting the variables involved in the crash and their values. We then perform backward analysis by leveraging record and replay and hardware watchpoints to locate the line or block of code that caused the crash.

Under the Hood: The first step of the analysis consists of recording a trace of the program execution with a crashing input. The trace includes snapshots of the contents of the memory and the registers of the recorded execution. After the program's execution is recorded, the execution is replayed and analyzed. The replay process supports moving between snapshots, thus creating an illusion of forward and reverse execution. Our engine leverages the reverse execution to track the data and control flows that lead to the program's crash. Starting at the crash location, we perform the following steps:

(1) *Initialize*: Analyze the crashing line, identify the variable(s) that caused the violation, and set watchpoint(s) on the identified variable(s).
(2) *Search*: Revert the program state to a previous snapshot in which the value of the observed variable has changed, *i.e.,* reverse execute until a watchpoint is hit. Identify the variables used to define the crashing variable(s) and validate the corresponding values by inspecting their memory mappings.
(3) *Decide*: If the observed variable changes state from invalid to valid (see Figure 3), then return the current line and the set of variables as the root cause. Otherwise return to step 1.

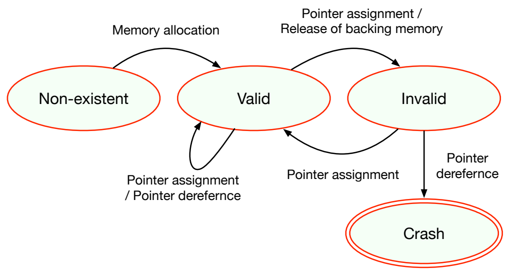

Memory allocation
Pointer assignment / Release of backing memory
Non-existent
Valid
Invalid
Pointer assignment / Pointer derefernce
Pointer assignment
Pointer derefernce
Crash

Figure 3: Pointer life-cycle

To assess our deduplication strategy, we examined how well it correctly identified the root cause for the applications shown in Table 1. Ground truth was obtained via a labor-intensive manual process. We found that our approach was able to identify multiple entry points as a single bug. A prominent example is *pcre*, where our engine identifies the bug and multiple entry points as a single entry. False positives occur with *tiff* and *libxml2* because the engine misattributes the root cause to incorrect source lines. Nonetheless, the

Table 1: Root Cause Analysis

Library	Bug Class	No. of bugs Actual	Ours	#Entry points
libarchive (v3.1.0)	Heap-buffer-overflow	8	5	9
libjpeg (v1.4.2)	Heap-buffer-overflow, Null Pointer Dereference	4	2	4
libplist (v1.11)	Heap-buffer-overflow Null Pointer Dereference	2	2	4
libpng (v1.2.56)	Null Pointer Dereference	1	1	2
libxml2 (v2.9.2)	Heap-buffer-overflow	4	6	7
pcre (v10.0)	Heap-buffer-overflow Null Pointer Dereference	5	5	15
tiff (v4.0)	Heap-buffer-overflow, Null Pointer Dereference	3	8	14
yaml (v0.5.3)	Logic error	1	1	1

results show our strategy can be used as an effective deduplication strategy.

During online deduplication, there were cases where we needed to revert to existing methods (*e.g., AddressSanitizer* [1]) when our engine failed to provide diagnostic output. That strategy appears to work well. For instance, in the case of CVE-2016-5102 (discussed in §5.3), the different fuzzers generated inputs that led to four distinct crash locations stemming from the same bug. Our core engine successfully identified the root cause for all discovered crash locations as a single bug, but other approaches (AddressSanitizer and stack backtrace methods) incorrectly identified the crashes as separate bugs. However, for CVE 2016-3186, our engine's output was inconclusive as it was unable to determine the root cause of the crashes. In that cause, we defaulted to the output from contemporary solutions.

3.4 Coverage Guided Fuzzing

With an understanding of how the first two stages of our framework operate, we now return to a discussion of three different strategies that can be employed at runtime.

Strategy 1 — Hunting similar bugs of the past: The success of this strategy hinges on the assumption that because the same coding practices may have been followed in the latest version of some code base, similar coding flaws may persist. To uncover the presence of such bugs, we first check if an input triggers a new code path. For example, AFL considers an input as interesting if it triggers new branch edges during execution. If that is the case, we validate whether this input is a quality input using the trained classifier. Only if both conditions are satisfied do we then allow the input to go on for mutation. Functionality wise, we are explicitly limiting the inputs the base fuzzer intended to pass on for mutation.

Strategy 2 — Hunting different bugs: Alternatively, it might be safe to assume that once bugs have been discovered in past versions of a program, the developers would have taken measures to minimize those errors in more current versions of the program. Hence, it may make sense to instead steer the fuzzing in a different direction. Strategy 2 does just that. First, we check if an input triggers new code paths. If so, we use the classifier to determine if the input is *dissimilar* to the ones seen in the past vulnerabilities, *i.e.,* if the input is classified as non-quality. If that condition holds, the input is selected for mutation. In this way, we are offering a more informative form of *path guidance* [10].

Strategy 3 — Dual mode: Modern fuzzers often get stuck in deep code paths or spend too much time on inefficient mutation operations that do not lead to entirely different code paths [4, 7, 16]. Consequently, they either do not generate quality inputs for a long time [27], or they end up finding quality inputs that trigger the same bug repeatedly. Both of these cases reduce the overall performance of a fuzzer. In the dual mode strategy, we allow for switching between ML-based heuristics, which can be triggered based on different conditions. For example, one may choose to switch heuristics if the fuzzer fails to generate any new crashing inputs for a certain period of time, or if the vast majority of recent crashes all have the same root cause. In this way, we allow the fuzzer to perform a more *comprehensive exploration* under fixed resource constraints.

4 IMPLEMENTATION DETAILS

From an engineering standpoint, the design of our framework consists of four modules (*Profiler, Feature Selector, Heuristic Generator, Vulnerability Explorer*) that map to the corresponding components outlined in Section 3. As an initial input to the workflow, we first collect quality and non-quality inputs from old versions of target programs we wish to fuzz. For that, we can use known crashing inputs to expedite the process or simply run the baseline fuzzer for some extended period on the old versions. Note that this is a one-time cost.

4.1 Profiler

As the name suggest, this module is responsible for recording the HPC events for all the code paths triggered by both quality and non-quality inputs during training. The trace, $\mathcal{T}$, for input, i, is given as $\mathcal{T}_i = \{e_{i1}, e_{i2}, \ldots, e_{i96}\}$, where e_{ij} represents the measurement of hardware events. Due to the limited number of programmable counters, to measure all 96 events, the Profiler executes the program with a specific input 24 times, each time configuring a different set of 4, non-overlapping, counters.

4.2 Feature Selector

The Selector is responsible for determining what specific hardware events should be used to guide the input selection process at runtime. First, the HPC traces are separated into two groups corresponding to quality and non-quality inputs. Since the number of traces for quality inputs will only be a small fraction of all the traces, the collection $\mathcal{S} = \mathcal{T}_1, \ldots, \mathcal{T}_N$, will be imbalanced. To mitigate any bias in the feature selection process, $\mathcal{S}$ is split into m smaller datasets by randomly selecting the same number of traces from each input class. Here, m is the ratio of non-quality to quality inputs.

Next, we use the *CfsSubsetEval* algorithm [20] to select the best subset of features. *CfsSubsetEval* evaluates the worth of a subset of features by considering the individual predictive ability of each feature along with the degree of redundancy between them. The evaluation is conducted using a 10-fold cross-validation approach. We then assign a score to each HPC event based on the number of times that event appears among the selected subset of features over all 10 runs. The overall score of an event is averaged across all m datasets. Informally, we denote that score as the information gain of an event. The coverage metric derived at the end of this process is $cov = \{e_1, e_2, e_3, e_4\}$ where e_j denotes the hardware events that meet some criteria (for example, the 4 events with the highest gain).

To further illustrate how this works, consider the analysis of a set of well-known libraries we would like to build coverage metrics for: libpng, libjpeg, yaml, tiff, pcre and libxml. In this case, the set $\mathcal{S}$ in Table 2 was derived by running the AFL fuzzer on known vulnerable versions of the target libraries for 48 hours each.

Table 2: Example training data

Programs	Quality inputs	Non-quality inputs
libarchive (v3.1.0) (bsdtar)	229	6417
libjpeg (v1.4.2) (cjpeg)	1136	52381
libplist (v1.11) (plist_test)	164	1544
libpng (v1.2.56) (pngtest)	496	9592
libxml2 (v2.9.2) (xmllint)	466	25489
pcre (v10.0) (pcre2test)	2533	42652
tiff (v4.0) (gif2tiff)	676	7595
yaml (v0.5.3) (parse)	682	11537

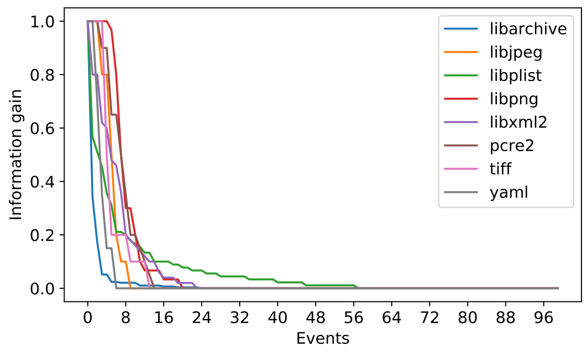


Figure 4: Information gain for sample libraries

Figure 4 shows the normalized information gain for all 96 events for each of the libraries. To further elucidate the relation between the selected events, we grouped the events into the 65 categories discussed in Section 3.1. Note that the gain varies across the benchmarks. This should be the case. While a few event classes show up across all benchmarks (*e.g.,* event classes *dtlb_load_misses, br_inst_retired, itlb_flush, offcore_requests*), different subsets of events have more discriminatory power for a given benchmark. For most of the programs, the gain is high for the top 10-12 events.

One can use this insight to build multiple coverage metrics for a given program. For example, we could select the best 4 events as a coverage metric or the next 4 events, depending on our desired outcome. Given that the cost of false positives and false negatives are very different in our approach, we ensured that the classifier built using the events has at least an F-measure value of 0.7 (shown in Figure 5). In addition, there must be sufficient number of events (*e.g.,* 6-8) with a high information gain to build multiple coverage metric sets. Based on these measures, we empirically set the information gain threshold at 0.4 — which yields the prerequisite 8 events in Figure 4 — and disregarding events with lower gain. Furthermore, when choosing the next-best grouping of events, we mandated that there was at least 50% difference in categories with the topmost 4 events. For example, in case of pcre, the topmost set would be: $\{e_{65}, e_{12}, e_{47}, e_{56}\}$, while the next set (above the cut-off) containing events from at least two different classes would be: $\{e_{46}, e_{12}, e_{54}, e_{41}\}$ from Table 8.

4.3 Runtime Heuristic

This component is responsible for deriving the heuristic that helps steer the input selection process. As noted in Section 3, we choose to use a multilayer perceptron approach because it offered good accuracy and was straightforward to translate the resulting classifier into a runtime heuristic. We used the Weka toolkit API to implement the MLP based classifier.

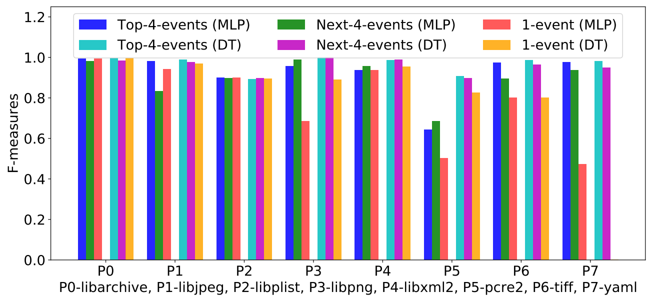


Figure 5: Comparison of F-measures

Figure 5 shows the performance of trained classifiers on a set of real-world libraries. We used 10-folds cross-validation approach to build our models. For the sake of comparison, we report the effectiveness of MLP and decision tree (DT) models built using the top 4 events, next 4 events and the event with the highest information gain for each program. Overall, the performance of MLP and DT classifiers are similar for most of the programs with 4 events, except in *pcre*. We choose MLP because DT uses discrete value of events which restricts the model, and in some cases the tree is quite complex with a large number of nested nodes (e.g., *pcre* has over 60 nodes). In the case where the models are built using the single event with the highest gain, the trained MLP model performed worse, and sometimes failed to identify the quality inputs.

4.4 Vulnerability Explorer

Motivated by the recent fuzzer benchmarking reports [45], we decided to apply our framework to AFL, MOpt [27] and Fairfuzz [24]. More details about this choice is given in §5.1. All of these fuzzers implement a *fork server* model. In fork server model, the fuzzed process goes through execve(), linking, libc initialization only once, and then clones from a stopped process image via copy-on-write.

The fork server stops at the first instrumented function to await commands from a fuzzer module. Our implementation is built in concert with the fork server model of a base fuzzer. Specifically, we modified base fuzzer's original instrumentation such that the child process waits to receive commands via a pipe before it can resume its execution. As it waits, our Profiler module configures the relevant HPC events for the child process created by the fork server. We use the `perf_event_open()` API to configure the HPC events. After the execution of the child process, the hardware events are recorded. Next, the recorded HPC values are used to build a feature vector, which is given as an input to the trained classifier. The classifier predicts whether the feature vector belongs to a quality class or non-quality class.

Table 3: Vulnerability Exploration Strategies

			HPC driven heuristics	
Mode	Objective	Strategies	Primary	Secondary
Single	Finding similar bugs	1-a	best 4 events	n/a
	Finding different bugs	2-a	best 4 events	n/a
	Finding different bugs	2-b	next 4 events	n/a
Dual	Finding similar bugs	3-a	best 4 events	next 4 events
	Finding different bugs	3-b	next 4 events	best 4 events

The vulnerability explorer module can be configured to pursue a myriad of strategies. Table 3 outlines variants of three strategies (see §3.4) we tested. In the single mode instantiation, we build the variants of Strategies 1 and 2 using either the best 4 events or the next 4 events, selected as per our criteria described in Section 4.2. In the dual mode case, we oscillate between the best 4 events and the next 4 events. The evaluation of these strategies in vulnerability discovery experiments is presented next.

5 EVALUATION

To assess the soundness of our approach, we first perform an evaluation on downstream versions of programs where ground truth (*i.e.,* vulnerabilities and patches) exists. Arguably, there is also practical relevance in fuzzing downstream versions of a program as doing so allows an analyst to investigate how long bugs of a particular type may have persisted. Additionally, practitioners may use such knowledge to direct their vulnerability discovery processes to hunt for bugs in prior versions where exploiting those bug may be easier.

Experimental Setup: All experiments were conducted in a virtualized environment running on a server consisting of Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz processor with 20 cores and 128 GB memory. Each virtual machine runs on a VMware hypervisor (that supports HPC monitoring), and is configured with 1 core (2 threads) and 8 GB of memory. Each fuzzer instance runs on a separate VM (to avoid hogging of resources by a particular instance), and each experimental run is conducted for a period of 24 hours. To reduce the impacts of randomness in the fuzzing process, we repeat each experiment 6 times [23, 28]. Our benchmark consists of utilities from the real-world libraries shown in Table 4.

5.1 On Extensibility

To decide which fuzzers to apply our framework to, we initially selected 4 fuzzers based on source code availability and findings from a recent fuzzer benchmarking report [45] that ranks fuzzers based on the medians of reached coverage on different benchmarks. Table 4 presents the results of these fuzzers in terms of bug discovery and their consistency over repeated runs. For some programs (*e.g.,* libjpeg), we found multiple instances of the same bug that roots to different code locations (shown in parentheses). Finding multiple occurrence of the same bug also adds to the efficacy of a fuzzer, therefore, we classify them as independent bugs although they were documented using same CVE. The results show that AFL, MOpt and Fairfuzz fall on the higher end of the spectrum in terms of consistency and finding more bugs, while Angora falls on the lower end. In addition, following the recommendation of Klees et al. [23], we note that recent works address the randomness in the fuzzing by conducting multiple runs and then present an aggregated result. But, from a practitioner's point of view, a true measure of the quality of a fuzzer is whether it can consistently find a particular bug in repeated runs. Along those lines, we also report the consistency of an approach to find a specific bug over multiple runs.

Based on Google's evaluation using the critical distance metric [45], AFL, MOpt and Fairfuzz performed well on their fuzz benchmarks. We decided to apply our framework to AFL as it is by far the most popular fuzzer. MOpt targets a different point in the fuzzing space in that it offers a novel mutational scheduling scheme to better enable mutational-fuzzers to discover vulnerabilities. For that reason, we also selected MOpt as a candidate to apply our framework to. Lastly, we also extended Fairfuzz for a different point in the space. We chose Fairfuzz because it uses a unique seed selection approach for guiding the fuzzer toward rarely executed paths.

5.2 On Expediency

One way to measure whether OmniFuzz finds potential bugs faster is to compare the time-to-crash ratio for the crashes that are common between the approaches. That metric has been used elsewhere [39, 54] as, at first blush, it appears to be a decent measure of the efficiency of a fuzzer in reaching bugs. If we apply that yardstick, the results show that we can find crashes much faster than the base approach: 1200X faster in the *tiff* benchmark using strategy 1-a, 70x faster in *yaml* using strategy 2-a, and 40X faster in *libjpeg* using strategy 2-a. While impressive, the time-to-crash ratio is somewhat misleading because (from a practitioner's standpoint) the difference between finding crashes in seconds versus minutes may not be that suasive. Therefore, we take a slightly different approach and report the speedup relative to a predefined granularity. E.g., if the granularity is 15 minutes, and a crash is obtained in 30 seconds by one fuzzer and in 14 minutes by the other, we consider the speedup as 1.

Figures 6a-6c show the speedup for each unique crash found on a per run basis for AFL, MOpt, and Fairfuzz respectively. For a more detailed look, we refer the reader to Tables 9-11 in the Appendix. In our analyses, the uniqueness of a crash is measured by the *stack trace* approach [23]. A larger ratio means that our

Table 4: Consistency in finding bugs *(over 6 runs)*

Programs	CVE/Bugs	Bug-description	Bug ID	AFL	Mopt	Angora	Fairfuzz
libarchive (v3.0.3)	CVE-2015-8932	Heap-buffer-overflow	1	6	☑	⬇	⬇
	ID not assigned	Heap-buffer-overflow	2	6	☑	⬇	⬇
	CVE-2015-8928	Heap-buffer-overflow	3	2	⬇	⬆	⬆
	CVE-2015-8923	Out of bounds read	4	0	☑	☑	☑
	ID not assigned	Out of bounds read	5	0	☑	☑	☑
libjpeg (v1.3.0)	CVE-2018-11212	Divide-by-zero	6	5	⬆	⬆	☑
	CVE-2018-11213	Heap-buffer-overflow$_{(1)}$	7	5	⬆	⬇	⬇
		Heap-buffer-overflow$_{(2)}$	8	3	⬇	⬇	⬇
		Heap-buffer-overflow$_{(3)}$	9	3	⬇	⬇	⬇
		Heap-buffer-overflow$_{(4)}$	10	3	⬇	⬇	⬇
		Heap-buffer-overflow$_{(5)}$	11	2	⬇	⬇	⬇
	CVE-2018-11214	Heap-buffer-overflow$_{(1)}$	12	5	⬆	⬇	⬇
		Heap-buffer-overflow$_{(2)}$	13	5	☑	⬇	⬇
		Heap-buffer-overflow$_{(3)}$	14	0	⬆	⬆	☑
libplist (v1.9)	CVE-2017-5209	Type conversion	15	6	☑	✖	☑
	ID not assigned	Out of bounds read	16	6	☑	✖	☑
libpng (v1.0.69)	ID not assigned	Null Pointer Dereference	17	3	⬇	⬇	⬇
libxml2 (v2.8.0)	CVE-2017-9049/9050	Heap-buffer-overflow	18	1	⬇	✖	⬆
	CVE-2016-1835	Heap use-after-free	19	0	☑	✖	☑
	CVE-2015-7497	Heap-buffer-overflow	20	0	☑	✖	⬆
	CVE-2015-7498	Heap-buffer-overflow	21	4	⬇	✖	☑
pcre (v8.38)	Bug #1783	Out of bounds read	22	3	⬇	⬇	⬇
	CVE-2017-11164	Stack overflow	23	0	⬆	⬆	☑
tiff (v3.7.0)	CVE-2016-5102	Heap-buffer-overflow	24	5	⬇	⬇	⬆
	CVE-2016-3186	Heap-buffer-overflow	25	6	⬇	⬇	☑
	CVE-2013-4244	Heap-buffer-overflow	26	5	⬇	⬆	⬆
	CVE-2013-4231	Heap-buffer-overflow	27	5	⬇	⬆	⬆
	ID not assigned	Heap-buffer-overflow	28	5	⬇	⬇	⬆
yaml (v0.3.0)	ID not assigned	Logic Error	29	6	☑	✖	☑

Numbers inside parentheses denote the distinct instances of bugs with different root causes, but addressed by a single CVE. Benchmarks that Angora failed to run on are listed as ✖. The number of times a bug is found by a fuzzer is referenced with respect to AFL and shown as ☑ to denote the same as AFL, ⬆ more than AFL, and ⬇ fewer than AFL.

approach finds the bug in less time. We apply the pairwise two-tailed Mann-Whitney U approach to test for statistical significance. The results show that for most of the programs, the time-to-crash data obtained using our approach are *statistically different* from the base fuzzer. Specifically, we induce crashes *3.5X* faster in *libjpeg*, *7X* faster in *tiff*, and *2X* faster in *yaml* and *libarchive* when AFL is the base fuzzer. We perform worse in the case of *libxml2*, although there are only a handful of crashes for that benchmark. With MOpt as the base fuzzer, we improve the time needed to find unique crashes by *1.6X* in *libjpeg*, *3X* in *tiff*, *2.3X* in *yaml*, *1.1X* in *libarchive*. The dual mode (strategies 3-a, 3-b) outperforms the single mode in the majority of cases. Finally, in the case of Fairfuzz, our approach improves the time to finding unique crashes by *3.0X* in *libarchive* and *libxml2*, *2.3X* in *tiff* and *3.4X* in *yaml*.

That said, our approach performs poorly in some cases. Our painstaking analysis showed that the benefits of our extension are undermined under certain circumstances. First, if there are only a few initial seed inputs and those seeds are not selected by our heuristic, then the fuzzer's progress slows down. For instance, *libarchive, tiff* and *libxml2* have only one seed input. With our strategies such as 1-a and 3-a, the process gets stuck waiting for the mutation to generate entirely different inputs that can stir the fuzzing exploration in a new direction. This acts as a chokepoint and impedes progress. This limitation can be addressed by starting off with large and diverse initial seeds (*e.g.,* as was the case with *yaml*).

Second, we found that the full benefits offered by our extension may not be realized when composed with certain baseline fuzzers.

This is especially true when their core guiding technique(s) counteracts that of our HPC based models. For example, in the case of Fairfuzz, we found that strategies 1-a and 3-a do not work well because of Fairfuzz's conservative policy, which heavily trims inputs based entirely on rare branches. Given that our approach leverages more contextual coverage information obtained through HPC, the restrictions imposed by Fairfuzz undercuts these potential benefits.

5.3 On Bug Discovery

For bug analysis, first we deduplicated crashes based on function names, line numbers and *crashing cause* as reported by *AddressSanitizer* [49] or manually using a debugger (*e.g., gdb*). Next, we manually inspected all unique crashes to identify their *root cause*, and classify them as a unique bug based on the root cause. We supplemented our understanding of each bug with the publicly available CVEs, bug reports and the developers' patches. The extra validation is important as the root cause can be different from the crashing cause given by *AddressSanitizer* [1], *e.g.,* in libplist a type conversion bug led to a heap overflow vulnerability (CVE-2017-5209), while in libtiff a buffer overflow (CVE 2016-5102) led to overwriting of two different pointers, resulting in two distinct crashes — invalid read access to a file pointer, and invalid free of memory.

We present our evaluation on extending AFL, MOpt and Fairfuzz base fuzzers using our approach in Table 5. Overall, our approach has similar or higher consistency for finding bugs. In particular, with our extension, AFL is able to find 3 more bugs, MOpt's discovery improves by 5 bugs, while Fairfuzz by 8 bugs. This amounts to $> 13\%$ improvement on AFL, $> 29\%$ on MOpt and $> 53\%$ on Fairfuzz. These results aptly demonstrate the effectiveness of our approach in improving bug discovery.

5.4 On Comprehensiveness

Lastly, we explore the effectiveness of a portfolio configuration [17] where strategies 2-b, 3-a and 3-b are run independently, and the results combined. For fairness, the baseline fuzzer is run 18 times, while our strategies are run 6 times each. Table 6 compares base fuzzers with the portfolio configuration measured by the consistency of bugs. Notice that the combined strategies led to more bugs than the base, *i.e.,* 2 more bugs in AFL, 5 more bugs in MOpt, and 2 more bugs in Fairfuzz. Moreover, our approach has higher consistency in finding the bugs than the base fuzzers. Interestingly, our strategies explore 62% of the base paths on average (see Table 12 in the Appendix). Thus, our approach can be considered more directed in the search for bugs.

5.5 Vulnerability Discovery in the Wild

Satisfied with the performance of Omnifuzz, we decided to use it to fuzz the current versions of *libjpeg*, *libarchive* and *pcre*. We limited fuzzing to those three libraries due to time and resource constraints. Within several hours, we induced a number of crashes that mapped to 9 new bugs. The discovered bugs are listed in Table 7. After reporting the vulnerabilities, 2 CVEs were assigned, and another two bugs were immediately fixed. Note that these are heavily fuzzed programs, and are continuously fuzzed on a large scale resources, such as by Google's continuous fuzzing framework for open source software. For a few others, the maintainers argued that the bugs identified were due to specific features (*e.g.,* recursion - Bug-2484), and it is up to the programmer to ensure correct inputs are used (*e.g.,* Bug-2479-2483).

6 LIMITATIONS

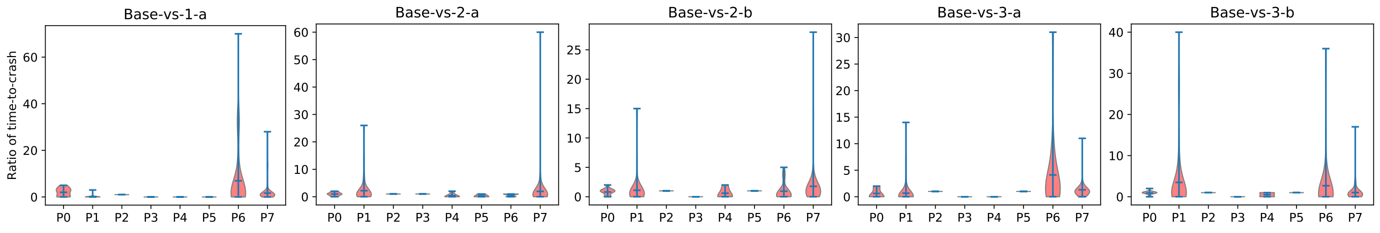
Clearly, the approach we advocate in this paper is not ideal for all fuzzing campaigns. First, our approach requires a priori knowledge of prior bugs in past versions to be able to guide the fuzzing process. That said, it is possible that techniques from transfer learning [18] and few-shot learning [55] can be used to build more sophisticated models when a sufficient number of quality inputs are not readily available. Moreover, we are not arguing that the approach we take using multi-layer perceptrons to build our models is the best choice. As stated earlier, we chose that solution because it offered significant operational benefits. Secondly, though we make no assumptions about the input scheduling algorithm used, we do not study how the guidance we give during the input selection process could impact optimality. The theoretical frameworks and formalizations by Rebert et al. [44] and Hayes et al. [21] could help in that regard. Lastly, the models we build may not be robust against anti-fuzzing techniques [19, 22] that try to impede automated bug finding.

Our deduplication technique is, at present, not applicable to fuzzing at the binary-level as our current instantiation mandates that we have access to source code of the target program, and that it be compiled with debugging symbols and no optimizations. While this is not an issue for open-source software, closed-source applications are also subject to fuzzing.

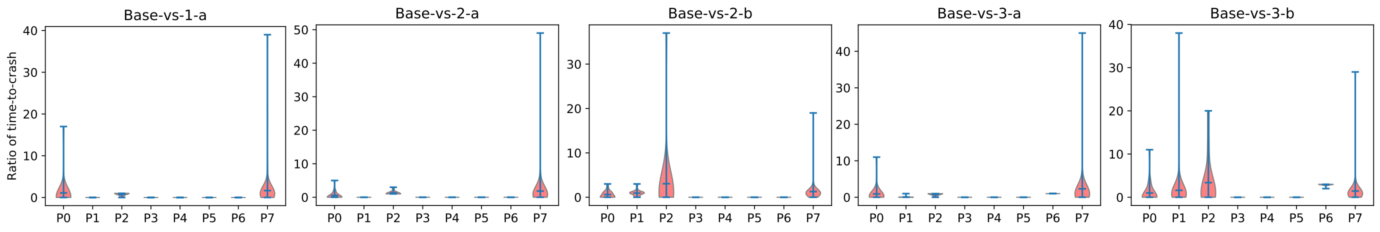
7 RELATED WORK

Deduplication: To date, both deduplication and root cause analysis have been active areas of research. From an industry standpoint, stack backtrace hashing and edge coverage are the most common approaches to deduplication [13, 23, 28]. However, these approaches suffer from either over-approximation or under-approximation [23, 28]. To address those limitations, several academic solutions have been proposed. For example, Lin et al. [26] used static and dynamic analysis at the source code level to detect and determine the root cause of out-of-bounds vulnerabilities. Cui et al. [12] proposed to deduplicate crashes in production systems by reconstructing dataflow from a core dump, and performing backward analysis from the crash point. Crashes are deduplicated based on the first function from which the bad value that caused the crash was derived. Xu et al. [58] proposed to improve root cause analysis when a core dump contains corrupted data (*e.g.,* due to memory corruption vulnerabilities). Xu et al. [59] later proposed an approach (and extensions [36]) that uses the Intel processor trace feature and a core dump to perform offline binary analysis to recover instructions that lead to a crash. Subsequently, Cui et al. [11] suggested a refinement wherein the accuracy of the recovered data flow is improved.

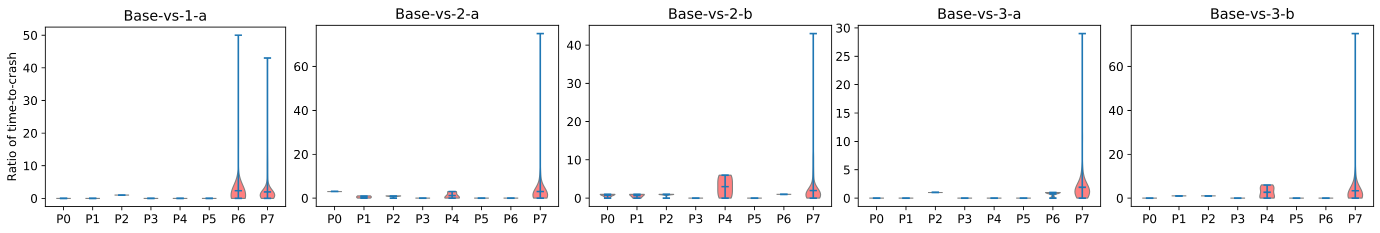
None of these approaches were designed to be used for online deduplication. We incorporated several ideas (*e.g.,* backward dataflow [12] and record & replay [8]) from these works in the design of our deduplication strategy for guiding the fuzzing process.



(a) On AFL



(b) On Mopt.



(c) On Fairfuzz.

Figure 6: Relative time to find a crash. P0: libarchive, P1: libjpeg, P2: libplist, P3: libpng, P4: libxml2, P5: pcre, P6 :tiff, P7: yaml. A larger ratio in the violin plots means that our approach finds crashes quicker. Statistical significance via a pairwise two-tailed test is presented in Tables 9-11 in Appendix A.2.

Fuzzing: The art and science of fuzzing has witnessed explosive growth [3, 4, 6, 27, 41–43, 53, 60], driven in part by the booming software security market. Many approaches [4, 43] try to explore low-frequency paths in order to reach bugs hidden inside less explored paths. Vuzzer [43], for example, uses control and data-flow features to prioritize deep and less frequently explored code paths. It performs taint analysis to infer the data types at certain offsets in the input, and then uses that knowledge to mutate inputs. CollAFL [16] prioritizes input selection based on more untouched branches (to increase the coverage) as well as more memory access operations (to find memory corruptions). Similarly, Angora [7] seeks to increase branch coverage by solving path constraints using context-sensitive branch count and byte-level taint tracking. To distinguish the executions of the same branch in different contexts, Angora appends context to the branch IDs to explore paths more pervasively. By tracking which input bytes flow into each path constraint, the approach mutates only these bytes instead of the entire input.

Li et al. [25] applied a static approach on known vulnerable programs to extract basic block information, comprising the number of call instruction, operand types and string types. In some sense, these attributes can be viewed as characterizing program behavior, albeit not at the architectural level. Based on these attributes, a deep learning model is built wherein scores are assigned to basic blocks traversed by the program. By calculating scores for basic blocks, Li et al. [25] infers the fitness of an input for mutation during fuzzing. We infer similar control flow and data flow behavior information using performance counter events but with low overhead.

Concurrent to our work, Österlund et al. [39] proposed a sanitizer-based approach to direct fuzzing towards triggering sanitizer checks to find bugs faster. At a high-level, they also prioritize paths by steering the program towards locations that are more likely to have bugs. Unlike ours, their approach is based on sanitizer checks, which may miss certain bug types *e.g.,* logical errors. Moreover, we implement a feedback mechanism to actively guide fuzzing and switch between multiple strategies at runtime.

A handful of works [44, 53, 57] have examined what scheduling algorithms produce the best results for seed selection. Rebert et al. [44] formalize the notion of ex post facto optimality seed selection, and provide evidence-driven techniques for identifying the quality of a seed selection strategy compared to an optimal solution. Unlike our work, these proposals focus on ways to measure the optimal case for bugs found with a particular subset of seeds or to find a “good” set of seeds that can be reused from one application to another. Overall, these works show that the choice of seed scheduling algorithm can significantly impact the success of fuzzing campaigns.

Table 5: Performance of *OmniFuzz vs base* measured by consistency of finding bugs *(over 6 runs)*

Programs	Bug ID	AFL	1-a	2-a	2-b	3-a	3-b	MOpt	1-a	2-a	2-b	3-a	3-b	Fairfuzz	1-a	2-a	2-b	3-a	3-b
libarchive	1	6	⬇	☑	☑	⬇	☑	6	⬇	⬇	☑	☑	☑	2	⬇	⬇	⬆	⬇	⬇
	2	6	⬇	⬇	☑	⬇	☑	6	☑	☑	⬇	☑	⬇	1	⬇	⬇	⬇	⬇	☑
	3	2	⬆	⬆	⬆	⬆	⬇	0	⬆	⬆	⬆	⬆	⬆	6	⬇	⬇	⬇	⬇	⬇
	4	0	☑	☑	☑	☑	☑	0	☑	☑	☑	☑	☑	0	☑	☑	☑	☑	☑
	5	0	☑	☑	☑	☑	☑	0	⬆	⬆	⬆	⬆	⬆	0	☑	☑	☑	☑	☑
libjpeg	6	5	⬇	⬆	⬆	⬆	⬆	6	☑	☑	☑	☑	☑	5	⬇	⬇	⬇	⬇	⬇
	7	5	☑	⬆	⬆	⬆	⬆	6	☑	⬇	☑	☑	☑	0	☑	⬆	⬆	⬆	⬆
	8	3	⬇	☑	☑	⬇	☑	0	☑	☑	☑	☑	☑	0	☑	⬆	⬆	⬆	⬆
	9	3	⬇	☑	☑	☑	☑	0	☑	☑	☑	☑	☑	0	☑	⬆	⬆	⬆	⬆
	10	3	⬇	☑	☑	⬇	☑	0	☑	☑	☑	☑	☑	0	☑	⬆	⬆	⬆	⬆
	11	2	☑	⬆	⬆	⬆	⬆	0	☑	☑	☑	☑	☑	0	☑	⬆	⬆	⬆	⬆
	12	5	☑	⬆	⬆	☑	⬆	6	☑	⬇	☑	⬇	☑	0	☑	⬆	⬆	⬆	⬆
	13	5	⬇	⬇	⬆	⬇	☑	5	⬇	⬇	⬆	⬇	☑	0	☑	⬆	☑	☑	⬆
	14	0	⬆	⬆	⬆	⬆	⬆	3	⬆	⬇	⬆	⬇	⬆	0	☑	⬆	☑	⬆	⬆
libplist	15	6	☑	☑	☑	☑	☑	6	☑	☑	☑	☑	⬇	6	☑	☑	☑	☑	⬇
	16	6	☑	☑	☑	☑	☑	6	☑	☑	☑	☑	⬇	6	☑	☑	☑	☑	⬇
libpng	17	3	☑	☑	☑	☑	☑	0	☑	☑	☑	☑	☑	0	☑	☑	☑	☑	☑
libxml2	18	1	⬇	⬇	⬆	⬇	⬆	0	☑	☑	⬆	☑	⬆	3	⬇	☑	☑	⬇	☑
	19	0	☑	☑	☑	⬆	☑	0	☑	☑	☑	☑	☑	0	☑	☑	☑	☑	☑
	20	0	⬆	⬆	⬆	⬆	☑	0	⬆	⬆	⬆	⬆	⬆	5	⬇	⬇	⬇	⬇	⬇
	21	4	⬇	⬇	⬇	⬇	⬇	0	☑	☑	☑	☑	⬆	4	⬇	⬇	⬇	⬇	⬇
pcre	22	3	⬇	☑	☑	☑	☑	1	⬆	⬆	⬆	⬆	⬆	0	☑	☑	☑	☑	☑
	23	0	☑	☑	☑	☑	☑	1	⬇	⬆	⬇	☑	☑	0	☑	☑	☑	☑	☑
tiff	24	5	⬆	⬇	⬆	⬆	☑	2	⬇	☑	⬆	☑	☑	6	☑	⬇	⬇	⬇	⬇
	25	6	☑	☑	☑	☑	☑	2	⬇	☑	⬆	☑	☑	6	☑	⬇	⬇	⬇	⬇
	26	5	⬆	⬇	☑	⬆	☑	2	⬇	☑	⬆	☑	☑	6	☑	⬇	⬇	⬇	⬇
	27	5	⬆	⬇	⬆	⬆	☑	2	⬇	☑	⬆	☑	☑	6	☑	⬇	⬇	⬇	⬇
	28	5	⬆	⬇	⬆	⬆	☑	2	⬇	☑	⬆	⬆	☑	6	☑	⬇	⬇	⬇	⬇
yaml	29	6	☑	☑	☑	☑	☑	6	☑	☑	☑	☑	☑	6	☑	☑	☑	☑	☑

The number of times a bug is found by an approach is presented with respect to the base fuzzer and is shown as ☑ to denote same as the base fuzzer, ⬆ more than the base fuzzer, and ⬇ fewer than the base fuzzer.

The strategies we apply to accept or reject inputs to be queued for mutation are complementary to the scheduling algorithms.

Hardware assistance: Lastly, there is a growing body of research on hardware-assisted fuzzing [10, 15, 47, 50]. These approaches use the processor trace feature to gather information for gauging program coverage. Although these approaches leverage processor trace facilities to efficiently collect an execution trace, the underlying coverage-guiding principle is similar to that of AFL.

8 CONCLUSION

We demonstrate inefficiencies in contemporary coverage-guided fuzzers, due principally to their equal prioritization of all program paths. To address the inefficiencies, we propose a framework called *OmniFuzz* that incorporates on-the-fly crash deduplication as a feedback mechanism to coax the fuzzer to change course when no unique crashes are obtained for some time. A unique aspect of *OmniFuzz* is its use of performance counter data to derive information that can be used as a coverage metric to guide input selection. Armed with these capabilities, we show how one can devise a multitude of strategies to guide a fuzzer toward finding similar or different bugs from those discovered in the past. These improvements can be integrated with most of contemporary fuzzers as they are not tied to a particular architecture. Our experimental results show that *OmniFuzz* can find more unique bugs, and can also find bugs significantly faster than the base fuzzer it augments (*e.g.,* AFL, MOpt, Fairfuzz). Taken as a whole, our experiments aptly demonstrate that our vulnerability-aware selection of coverage metrics, coupled with our on-the-fly deduplication technique, offers an expedient and comprehensive solution for improving the performance of a base fuzzer.

ACKNOWLEDGMENTS

We thank Prof. Yousra Aafer and the anonymous reviewers for their suggestions on how to improve the paper. We also thank Murray Anderegg for his assistance with deploying the infrastructure used in this study. This work was supported in part by the Department of Defense (DoD) under award FA8750-17-C-0016 and the National Science Foundation (NSF) under award CNS-1749895. Any opinions,

Table 6: Portfolio mode consistency *(over 18 runs)*

Programs	Bug ID	AFL	OmniFuzz	MOpt	OmniFuzz	Fairfuzz	OmniFuzz
libarchive	1	18	⬇	18	✓	11	⬇
	2	17	⬇	17	⬇	2	⬇
	3	2	⬆	0	⬆	18	⬇
	4	0	✓	0	✓	1	⬇
	5	0	✓	0	⬆	2	⬇
libjpeg	6	17	⬆	18	✓	15	⬇
	7	17	⬆	18	✓	2	⬆
	8	5	⬆	2	⬇	0	⬆
	9	5	⬆	2	⬇	0	⬆
	10	5	⬆	2	⬇	0	⬆
	11	4	⬆	2	⬇	0	⬆
	12	17	✓	18	⬇	4	⬆
	13	15	✓	17	⬇	4	⬇
	14	11	⬆	15	⬇	4	⬇
libplist	15	18	✓	18	⬇	18	⬇
	16	18	✓	18	⬇	18	⬇
libpng	17	3	⬆	0	✓	0	✓
libxml2	18	1	⬆	0	⬆	10	⬇
	19	0	⬆	0	✓	0	✓
	20	0	⬆	0	⬆	11	⬇
	21	4	⬆	0	⬆	13	⬇
pcre	22	3	⬆	1	⬆	0	✓
	23	0	✓	2	✓	0	✓
tiff	24	17	✓	2	⬆	18	⬇
	25	18	✓	2	⬆	18	⬇
	26	17	⬇	2	⬆	18	⬇
	27	17	✓	2	⬆	18	⬇
	28	17	✓	2	⬆	18	⬇
yaml	29	6	⬆	8	⬆	18	✓

The number of times a bug is found by an approach is presented with respect to the base fuzzer and shown as ✓ to denote same as the base fuzzer, ⬆ more than the base fuzzer, and ⬇ fewer than the base fuzzer.

Table 7: List of new bugs discovered by our approach

Programs	Versions	CVEs/Bugs	Bug details
libjpeg-turbo (cjpeg)	2.0.4	CVE-2020-13790	Heap-based buffer-over-read in get_rgb_row()
libarchive (bsdtar)	3.4.0, 3.4.1dev	CVE-2019-19221	Out-of-bounds read in archive_wstring_append_from_mbs()
	3.4.1dev	Bug 1298	Out-of-bounds write in archive_string_append_from_wcs()
pcre (pcre2test)	10.34-RC1, 10.33	Bug-2479	Heap overflow in GETCHARINC()
		Bug-2480	Heap overflow in GETCHARLEN()
		Bug-2481	Heap overflow in GETCHARLENTEST()
		Bug-2482	Heap overflow in GETCHARINCTEST()
		Bug-2483	Out-of-bounds read in internal_dfa_match()
		Bug-2484	Stack-overflow in internal_dfa_match()

findings, and conclusions expressed herein are those of the authors and do not necessarily reflect the views of the DoD or NSF.

REFERENCES

[1] 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 *USENIX Security Symposium.* 235–252.

[2] Marcel Böhme, Valentin Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Symposium on the Foundations of Software Engineering (ESEC/FSE).* 1–11.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *ACM Conference on Computer and Communications Security.* 2329–2344.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *ACM Conference on Computer and Communications Security.* 1032–1043.

[5] Augusto Born de Oliveira. 2015. Measuring and Predicting Computer Software Performance: Tools and Approaches. http://hdl.handle.net/10012/9259

[6] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *ACM Conference on Computer and Communications Security.* 2095–2108.

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

[8] Yue Chen, Mustakimur Khandaker, and Zhi Wang. 2017. Pinpointing Vulnerabilities. In *ACM Asia Conference on Computer and Communications Security.* 334–345.

[9] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *IEEE Symposium on Security & Privacy.*

[10] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. PTrix: Efficient Hardware-assisted Fuzzing for COTS Binary. In *ACM Asia Conference on Computer and Communications Security.* ACM, 633–645.

[11] 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 *USENIX Symposium on Operating Systems Design and Implementation.* 17–32.

[12] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. 2016. Retracer: Triaging Crashes by Reverse Execution From Partial Memory Dumps. In *IEEE/ACM International Conference on Software Engineering.* 820–831.

[13] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *IEEE/ACM International Conference on Software Engineering.* 1084–1093.

[14] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In *IEEE Symposium on Security & Privacy.* 20–38.

[15] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. 2020. PHMon: A Programmable Hardware Monitor and Its Security Use Cases. In *USENIX Security Symposium.*

[16] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security & Privacy.* 679–696.

[17] Patrice Godefroid. 2020. Fuzzing: Hack, Art, and Science. In *Communications of the ACM*, Vol. 63. 70–76.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2019. *Deep Learning.* MIT Press.

[19] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. 2019. Anti-Fuzz: Impeding Fuzzing Audits of Binary Executables. In *USENIX Security Symposium.* 1931–1947.

[20] Mark Andrew Hall. 1999. Correlation-based Feature Selection for Machine Learning. (1999).

[21] Liam Hayes, Hendra Gunadi, Adrian Herrera, Jonathon Milford, Shane Magrath, Maggi Sebastian, Michael Norrish, and Antony L Hosking. 2019. MoonLight: Effective Fuzzing with Near-Optimal Corpus Distillation. *arXiv preprint arXiv:1905.13055* (2019).

[22] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. 2019. Fuzzification: Anti-Fuzzing Techniques. In *USENIX Security Symposium.* 1913–1930.

[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security.* 2123–2138.

[24] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *ACM/IEEE International Conference on Automated Software Engineering.*

[25] Yuwei Li, Shouling Ji, Chenyang Lv, Yuan Chen, Jianhai Chen, Qinchen Gu, and Chunming Wu. 2019. V-Fuzz: Vulnerability-Oriented Evolutionary Fuzzing. *arXiv preprint arXiv:1901.01142.*

[26] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. 2007. AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair. In *ACM Asia Conference on Computer and Communications Security.* 329–340.

[27] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium.* 1949–1966.

[28] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering.*

[29] Timothy Merrifield, Joseph Devietti, and Jakob Eriksson. 2015. High-Performance Determinism with Total Store Order Consistency. In *European Conference on Computer Systems.* 1–13.

[30] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. In *Communications of the ACM*, Vol. 33. ACM New York, NY, USA, 32–44.

[31] Charlie Miller. 2010. Babysitting an Army of Monkeys. In *CanSecWest.*

[32] David Molnar and Lars Opstad. 2010. Effective fuzzing strategies. In *CERT vulnerability discovery workshop*.

[33] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. In *ACM Computing Surveys*. 17:1–17:24.

[34] Tipp Moseley, Neil Vachharajani, and William Jalby. 2011. Hardware Performance Monitoring for the Rest of Us: A Position and Survey. In *International Conference on Network and Parallel Computing*. 293–312.

[35] Mozilla. 2018. Mozilla Record & Replay. https://rr-project.org/.

[36] Dongliang Mu, Yunlan Du, Jianhao Xu, Jun Xu, Xinyu Xing, Bing Mao, and Peng Liu. 2019. POMP++: Facilitating Postmortem Program Diagnosis with Value-set Analysis. *IEEE Transactions on Software Engineering*.

[37] Syed Shariyar Murtaza, Wael Khreich, Abdelwahab Hamou-Lhadj, and Ayse Basar Bener. 2016. Mining Trends and Patterns of Software Vulnerabilities. *Journal of Systems and Software* 117, 218–228.

[38] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. 2015. Establishing a Base of Trust with Performance Counters for Enterprise Workloads. In *USENIX Annual Technical Conference*. 541–548.

[39] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security Symposium*.

[40] Andy Ozment and Stuart E Schechter. 2006. Milk or Wine: Does Software Security Improve with Age?. In *USENIX Security Symposium*. 93–104.

[41] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *ACM Conference on Computer and Communications Security*. 2155–2168.

[42] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. In *arXiv preprint arXiv:1711.04596*.

[43] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security*, Vol. 17. 1–14.

[44] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *USENIX Security Symposium*. 861–875.

[45] FuzzBench: 2020-04-21 Report. 2020. URL: https://www.fuzzbench.com/reports /sample/index.html.

[46] Thomas Röhl, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2016. Validation of Hardware Events for Successful Performance Pattern Identification in High Performance Computing. In *Tools for High Performance Computing*. 17–28.

[47] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. KAFL: Hardware-assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*. 167–182.

[48] K Serebryany. 2015. LibFuzzer a Library for Coverage-guided Fuzz Testing. In *LLVM project*.

[49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*. 309–318.

[50] Robert Swiecki. 2016. Honggfuzz: A General-purpose, easy-to-use fuzzer with interesting analysis options. https://github.com/google/honggfuzz.

[51] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19, 6 (2014), 1665–1705.

[52] Sebastian Vogl and Claudia Eckert. 2012. Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture. In *European Workshop on System Security*.

[53] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven Seed Generation for Fuzzing. In *IEEE Symposium on Security & Privacy*. 579–594.

[54] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *International Symposium on Research in Attacks, Intrusions and Defenses*. 1–15.

[55] Yaqing Wang and Quanming Yao. 2019. Few-shot Learning: A Survey. arXiv:1904.05046 http://arxiv.org/abs/1904.05046

[56] Vince Weaver and Jack Dongarra. 2010. Can hardware performance counters produce expected, deterministic results?. In *Workshop on Functionality of Hardware Performance Monitoring*.

[57] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In *ACM Conference on Computer and Communications Security*. 511–522.

[58] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. 2016. CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump. In *ACM Conference on Computer and Communications Security*. 529–540.

[59] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts. In *USENIX Security Symposium*. 17–32.

[60] 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 *ACM Conference on Computer and Communications Security*. 2139–2154.

[61] Michal Zalewski. 2017. American Fuzzy Lop (AFL). *URL: http://lcamtuf.coredump.cx/afl.*

A APPENDIX

A.1 Hardware Events and Classes

We conservatively selected all those hardware performance counter events that could explain the high-level behavior of a program, but excluded those events that monitor low-level micro-architectural information or that are difficult to relate to high-level program behavior. For example, we excluded the events relating to the pipelining behavior of the CPU. In the end, we were left with a set of 96 events, which we further grouped into 65 classes, shown in Table 8. The criteria we used to group events were (a) events that are similar but only differ due to the change in the size of hardware components are considered as a single class, and (b) events that count hits instead of cycles are split into different classes.

A.2 Time-to-crash Analysis

Tables 9, 10 and 11 present a detailed analysis on speedup for each unique crash found on a per run basis on AFL, MOpt, and Fairfuzz respectively, as discussed in §5.2. The column 'E' represents the exact ratio of time to finding a crash by the base fuzzer vs our approach, while the column 'W' scales the relative speedup in terms of time window, where the granularity of the window is 15 minutes, i.e., if a fuzzer finds a crash in seconds vs 15 minute, W = 1. A larger ratio means that our approach finds the bug faster than the base. To test for statistical significance we apply the pairwise two-tailed Mann-Whitney U approach, represented by p-value (p). The results show that, for most of the programs, the time-to-crash data obtained using our approach are statistically different from the base fuzzer.

A.3 Paths Exploration

Table 12 shows that, on average, we explore less paths than the base fuzzer, but we still obtain as good, or better, outcomes. Thus, our approach can be considered more directed in the search for bugs.

Table 8: Hardware events and their classes

#	Classes	Events	#	Classes	Events
1	cache-references	cache-references	34	mem_load_retired_fb_hit	mem_load_retired.fb_hit
2	cache-misses	cache-misses	35	mem-stores	mem-stores
3	dTLB-loads	dTLB-loads	36	mem_inst_retired_all_stores	mem_inst_retired.all_stores
4	dtlb_load_misses	dTLB-load-misses	37	mem_inst_retired_split_stores	mem_inst_retired.split_stores
5	dtlb_load_misses_stlb_hit	dtlb_load_misses.stlb_hit	38	mem_inst_retired_stlb_miss_stores	mem_inst_retired.stlb_miss_stores
6	dtlb_load_miss_causes_a_walk	dtlb_load_misses.miss_causes_a_walk	39	mem_load_retired_l1_miss	mem_load_retired.l1_miss
7	dtlb_load_misses_walk_completed	dtlb_load_misses.walk_completed_1g dtlb_load_misses.walk_completed_2m_4m dtlb_load_misses.walk_completed_4k	40	mem_load_retired_l2_miss	mem_load_retired.l2_miss
8	dtlb_load_misses_walk_active_cycles	dtlb_load_misses.walk_active	41	mem_load_retired_l3_miss	mem_load_retired.l3_miss
9	dtlb_load_misses_walk_pending_cycles	dtlb_load_misses.walk_pending	42	instructions	instructions
10	dTLB-stores	dTLB-stores	43	arith.divider_active_cycles	arith.divider_active
11	dtlb_store_misses	dTLB-store-misses	44	branch-loads	branch-loads
12	dtlb_store_misses_stlb_hit	dtlb_store_misses.stlb_hit	45	br_inst_retired	branches, br_inst_retired.all_branches
13	dtlb_store_misses_miss_causes_a_walk	dtlb_store_misses.miss_causes_a_walk	46	br_inst_retired_conditional	br_inst_retired.conditional
14	dtlb_store_misses_walk_completed	dtlb_store_misses.walk_completed_1g dtlb_store_misses.walk_completed_2m_4m dtlb_store_misses.walk_completed_4k dtlb_store_misses.walk_completed	47	br_inst_retired_far_branch	br_inst_retired.far_branch
15	dtlb_store_misses_walk_active_cycles	dtlb_store_misses.walk_active	48	br_inst_retired_near_call	br_inst_retired.near_call
16	dtlb_store_misses_walk_pending_cycles	dtlb_store_misses.walk_pending	49	br_inst_retired_near_return	br_inst_retired.near_return
17	iTLB-loads	iTLB-loads	50	br_inst_retired_near_taken	br_inst_retired.near_taken
18	iTLB-load-misses	iTLB-load-misses	51	br_inst_retired_not_taken	br_inst_retired.not_taken
19	itlb_misses_stlb_hit	itlb_misses.stlb_hit	52	branch-load-misses	branch-load-misses
20	itlb_misses_causes_a_walk	itlb_misses.miss_causes_a_walk	53	br_misp_retired	branch-misses br_misp_retired.all_branches br_misp_retired.all_branches_pebs
21	itlb_misses_walk_completed	itlb_misses.walk_completed_1g itlb_misses.walk_completed_2m_4m itlb_misses.walk_completed_4k itlb_misses.walk_completed	54	br_misp_retired_conditional	br_misp_retired.conditional
22	itlb_misses_walk_active_cycles	itlb_misses.walk_active	55	br_misp_retired_near_call	br_misp_retired.near_call
23	itlb_misses_walk_pending_cycles	itlb_misses.walk_pending	56	br_misp_retired_near_taken	br_misp_retired.near_taken
24	L1-dcache-loads	L1-dcache-loads	57	fp_arith_inst_retired	fp_arith_inst_retired.128b_packed_double fp_arith_inst_retired.128b_packed_single fp_arith_inst_retired.256b_packed_double fp_arith_inst_retired.256b_packed_single fp_arith_inst_retired.scalar_double fp_arith_inst_retired.scalar_single
25	L1-dcache-stores	L1-dcache-stores	58	fp_assist.any	fp_assist.any
26	L1-dcache-load-misses	L1-dcache-load-misses	59	hw_interrupts	hw_interrupts.received
27	L1-icache-load-misses	L1-icache-load-misses	60	uops_executed.x87	uops_executed.x87
28	longest_lat_cache.miss	longest_lat_cache.miss	61	longest_lat_cache.reference	longest_lat_cache.reference
29	mem-loads	mem-loads	62	machine_clears	machine_clears.count machine_clears.memory_ordering machine_clears.smc
30	mem_inst_retired_all_loads	mem_inst_retired.all_loads	63	offcore_requests	offcore_requests.all_data_rd offcore_requests.all_requests offcore_requests_buffer.sq_full offcore_requests.demand_code_rd offcore_requests.demand_data_rd offcore_requests.demand_rfo offcore_requests_outstanding.all_data_rd offcore_requests_outstanding.demand_code_rd offcore_requests_outstanding.demand_data_rd offcore_response offcore_response.demand_code_rd.any_response
31	mem_inst_retired_lock_loads	mem_inst_retired.lock_loads	64	tlb_flush	tlb_flush.dtlb_thread, tlb_flush.stlb_any
32	mem_inst_retired_split_loads	mem_inst_retired.split_loads	65	itlb_flush	itlb.itlb_flush
33	mem_inst_retired_stlb_miss_loads	mem_inst_retired.stlb_miss_loads			

Table 9: Performance of *OmniFuzz vs AFL* measured by the time to find a unique crash (*on a per run-basis*)

B: No. of base fuzzer crashes, C: Common crashes, E: Exact ratio, W: Window ratio, p: MannWhitney U test p-value

		Strategy-1-a				Strategy-2-a				Strategy-2-b				Strategy-3-a				Strategy-3-b			
Programs	B	C	E	W	p	C	E	W	p	C	E	W	p	C	E	W	p	C	E	W	p
libarchive	24	10	**2.67**	**2.00**	0.69	16	1.58	1.00	0.01	21	1.31	0.81	0.15	8	1.33	0.63	0.10	19	1.33	0.89	0.02
libjpeg	122	67	0.26	0.10	0.00	80	**4.29**	**2.20**	0.34	72	**2.29**	**1.10**	0.00	60	0.87	0.67	0.00	83	**9.36**	**3.48**	0.94
libplist	25	22	1.18	1.00	0.88	24	1.29	1.00	0.97	24	1.27	1.00	0.70	24	0.74	1.00	0.97	24	0.76	1.00	0.70
libpng	3	0	n/a	n/a	0.00	3	1.13	1.00	0.03	2	0.90	0.00	0.39	0	n/a	n/a	0.00	1	0.80	0.00	0.11
libxml2	11	0	n/a	n/a	0.04	5	1.05	0.40	0.34	5	0.99	0.60	0.84	0	n/a	n/a	0.23	4	0.83	0.50	0.34
pcre	3	0	n/a	n/a	0.00	3	0.99	0.33	0.19	2	1.04	1.00	0.13	2	1.02	1.00	0.13	3	1.02	1.00	0.16
tiff	76	62	**76.91**	**6.94**	0.00	27	0.40	0.59	0.11	37	1.01	0.95	0.00	45	**56.56**	**4.11**	0.59	48	**3.30**	**2.69**	0.78
yaml	396	156	**2.03**	**1.67**	0.82	141	**2.70**	**1.96**	0.22	108	**2.28**	**1.77**	0.94	98	1.64	**1.31**	0.88	147	1.68	**1.01**	0.01

Statistically significant MannWhitney U test p-values (p) are highlighted $p < 0.15$. $p < 0.10$. $p < 0.05$.

Table 10: Performance of *OmniFuzz vs MOpt* measured by the time to find a unique crash (*on a per run-basis*)

B: No. of base fuzzer crashes, C: Common crashes, E: Exact ratio, W: Window ratio, p: MannWhitney U test p-value

		Strategy-1-a				Strategy-2-a				Strategy-2-b				Strategy-3-a				Strategy-3-b			
Programs	B	C	E	W	p	C	E	W	p	C	E	W	p	C	E	W	p	C	E	W	p
libarchive	30	21	**1.93**	**1.10**	0.00	23	0.78	0.52	0.00	18	0.87	0.61	0.00	23	1.76	0.91	0.00	23	1.61	1.00	0.07
libjpeg	85	68	0.07	0.00	0.00	20	0.06	0.00	0.00	65	**8.24**	0.94	0.21	38	0.05	0.03	0.00	69	**10.01**	**1.61**	0.02
libplist	28	8	**8.28**	0.75	0.46	7	**3.91**	**1.29**	0.16	19	**5.75**	**3.05**	0.47	8	**3.26**	0.75	0.67	13	**6.44**	**3.38**	0.74
libpng	0	0	n/a	n/a	0.00	0	n/a	n/a	0.00	0	n/a	n/a	0.00	0	n/a	n/a	0.00	0	n/a	n/a	0.00
libxml2	0	0	n/a	n/a	0.00	0	n/a	n/a	0.00	0	n/a	n/a	0.00	0	n/a	n/a	0.00	0	n/a	n/a	0.00
pcre	3	0	n/a	n/a	0.64	1	0.18	0.00	0.16	0	n/a	n/a	0.64	1	0.13	0.00	0.23	0	n/a	n/a	0.33
tiff	19	0	n/a	n/a	0.00	0	n/a	n/a	0.00	0	n/a	n/a	0.00	10	1.16	1.00	0.00	14	**3.67**	**2.93**	0.36
yaml	264	91	**4.65**	**1.66**	0.01	91	**10.55**	**1.85**	0.37	103	**2.33**	**1.28**	0.00	93	**4.98**	**2.31**	0.34	103	**2.57**	**1.45**	0.02

Statistically significant MannWhitney U test p-values (p) are highlighted $p < 0.15$. $p < 0.10$. $p < 0.05$.

Table 11: Performance of *OmniFuzz vs Fairfuzz* measured by the time to find a unique crash (*on a per run-basis*)

B: No. of base fuzzer crashes, C: Common crashes, E: Exact ratio, W: Window ratio, p: MannWhitney U test p-value

		Strategy-1-a				Strategy-2-a				Strategy-2-b				Strategy-3-a				Strategy-3-b			
Programs	B	C	E	W	p	C	E	W	p	C	E	W	p	C	E	W	p	C	E	W	p
libarchive	13	0	n/a	n/a	n/a	1	**3.65**	**3.00**	0.11	4	**1.13**	0.75	0.68	0	n/a	n/a	n/a	0	n/a	n/a	n/a
libjpeg	5	0	n/a	n/a	n/a	2	0.08	0.50	0.16	3	0.34	0.67	0.13	0	n/a	n/a	n/a	3	0.50	1.00	0.37
libplist	20	8	0.49	1.00	0.57	13	0.45	0.92	0.02	14	0.84	0.93	0.25	12	0.93	1.00	0.47	10	0.48	1.00	0.29
libpng	0	0	n/a	n/a	n/a	0	n/a	n/a	n/a	0	n/a	n/a	n/a	0	n/a	n/a	n/a	0	n/a	n/a	n/a
libxml2	80	0	n/a	n/a	n/a	6	**1.74**	**1.17**	0.28	5	**3.79**	**3.00**	0.00	0	n/a	n/a	n/a	6	**3.16**	**2.67**	0.00
pcre	0	0	n/a	n/a	n/a	0	n/a	n/a	n/a	0	n/a	n/a	n/a	0	n/a	n/a	n/a	0	n/a	n/a	n/a
tiff	91	60	**20.29**	**2.35**	0.59	0	n/a	n/a	n/a	2	0.48	1.00	0.66	1	0.91	1.00	0.66	0	n/a	n/a	n/a
yaml	477	174	**2.74**	**1.98**	0.00	106	**4.09**	**2.99**	0.00	165	**2.71**	**1.98**	0.00	106	**2.78**	**1.91**	0.00	101	**4.23**	**3.38**	0.00

Statistically significant MannWhitney U test p-values (p) are highlighted $p < 0.15$. $p < 0.10$. $p < 0.05$.

Table 12: Paths explored

Programs	Base AFL	OmniAFL Portfolio	% of base paths	Base MOpt	OmniMOpt Portfolio	% of base paths	Base Fairfuzz	OmniFairfuzz Portfolio	% of base paths
libarchive	51634	31552	61.10	111157	40145	36.12	53000	15906	30.01
libjpeg	62314	49391	79.26	109165	41416	37.94	65111	24426	37.51
libplist	2435	2208	90.68	15364	3507	22.83	7096	3596	50.68
libpng	18699	21844	116.82	36386	21017	57.76	60200	6597	10.96
libxml2	115762	78503	67.81	126724	79175	62.48	103109	60918	59.08
pcre	81424	48114	59.09	222959	43614	19.56	73900	40115	54.28
tiff	10210	6208	60.80	1547	5362	346.61	10399	196	1.88
yaml	65787	25430	38.66	110981	18380	16.56	59768	15845	26.51
Combined			71.78			74.99			38.74