ParmeSan: Sanitizer-guided Greybox Fuzzing

Sebastian Österlund  
Vrije Universiteit  
Amsterdam

Kaveh Razavi  
Vrije Universiteit  
Amsterdam

Herbert Bos  
Vrije Universiteit  
Amsterdam

Cristiano Giuffrida  
Vrije Universiteit  
Amsterdam

Abstract

One of the key questions when fuzzing is where to look for vulnerabilities. Coverage-guided fuzzers indiscriminately optimize for covering as much code as possible given that bug coverage often correlates with code coverage. Since code coverage overapproximates bug coverage, this approach is less than ideal and may lead to non-trivial time-to-exposure (TTE) of bugs. Directed fuzzers try to address this problem by directing the fuzzer to a basic block with a potential vulnerability. This approach can greatly reduce the TTE for a specific bug, but such special-purpose fuzzers can then greatly underapproximate overall bug coverage.

In this paper, we present sanitizer-guided fuzzing, a new design point in this space that specifically optimizes for bug coverage. For this purpose, we make the key observation that while the instrumentation performed by existing software sanitizers are regularly used for detecting fuzzer-induced error conditions, they can further serve as a generic and effective mechanism to identify interesting basic blocks for guiding fuzzers. We present the design and implementation of ParmeSan, a new sanitizer-guided fuzzer that builds on this observation. We show that ParmeSan greatly reduces the TTE of real-world bugs, and finds bugs 37% faster than existing state-of-the-art coverage-based fuzzers (Angora) and 288% faster than directed fuzzers (AFLGo), while still covering the same set of bugs.

1 Introduction

Fuzzing is a common technique for automatically discovering bugs in programs. In finding bugs, many fuzzers try to cover as much code as possible in a given period of time [9, 36, 47]. The main intuition is that code coverage is strongly correlated with bug coverage. Unfortunately, code coverage is a huge overapproximation of bug coverage which means that a large amount of fuzzing time is spent covering many uninteresting code paths in the hope of getting lucky with a few that have bugs. Recent directed fuzzers [4, 8] try to address this problem by steering the program towards locations that are more likely to be affected by bugs [20, 23] (e.g., newly written or patched code, and API boundaries), but as a result, they underapproximate overall bug coverage.

We make a key observation that it is possible to detect many bugs at runtime using knowledge from compiler sanitizers—error detection frameworks that insert checks for a wide range of possible bugs (e.g., out-of-bounds accesses or integer overflows) in the target program. Existing fuzzers often use sanitizers mainly to improve bug detection and triaging [38]. Our intuition is that we can leverage them even more by improving our approximation of bug coverage in a target program. By applying directed fuzzing to actively guide the fuzzing process towards triggering sanitizer checks, we can trigger the same bugs as coverage-guided fuzzers while requiring less code coverage, resulting in a lower time-to-exposure (TTE) of bugs. Moreover, since compilers such as LLVM [25] ship with a number of sanitizers with different detection capabilities, we can steer the fuzzer either towards specific classes of bugs and behavior or general classes of errors, simply by selecting the appropriate sanitizers. For instance, TySan [14] checks can guide fuzzing towards very specific bugs (e.g., type confusion)—mimicking directed fuzzing but with implicitly specified targets—while ASan’s [37] pervasive checks can guide fuzzing towards more general classes of memory errors—mimicking coverage-guided fuzzing.

In this paper, we develop this insight to build ParmeSan, the first sanitizer-guided fuzzer. ParmeSan relies on off-the-shelf sanitizer checks to automatically maximize bug coverage for the target class of bugs. This allows ParmeSan to find bugs such as memory errors more efficiently and with lower TTE than existing solutions. Like coverage-guided fuzzers, ParmeSan does not limit itself to specific APIs or areas of the code, but rather aims to find these bugs, wherever they are. Unlike coverage-guided fuzzers, however, it does not do so by blindly covering all basic blocks in the program. Instead, directing the exploration to execution paths that matter—having the greatest chance of triggering bugs in
the shortest time.

To design and implement ParmeSan, we address a number of challenges. First, we need a way to automatically extract interesting targets from a given sanitizer. ParmeSan addresses this challenge by comparing a sanitizer-instrumented version of a program against the baseline to locate the sanitizer checks in a blackbox fashion and using pruning heuristics to weed out uninteresting checks (less likely to contain bugs). Second, we need a way to automatically construct a precise (interprocedural) control-flow graph (CFG) to direct fuzzing to the targets. Static CFG construction approaches are imprecise by nature [4] and, while sufficient for existing special-purpose direct fuzzers [4, 8], are unsuitable to reach the many checks placed by sanitizers all over the program. ParmeSan addresses this challenge by using an efficient and precise dynamically constructed CFG. Finally, we need a way to design a fuzzer on top of these building blocks. ParmeSan addresses this challenge by using a two-stage directed fuzzing strategy, where the fuzzer interleaves two stages (fuzzing for CFG construction with fuzzing for the target points) and exploits synergies between the two. For example, since data-flow analysis (DFA) is required for the first CFG construction stage, we use the available DFA information to speed up the second bug-finding stage. DFA-based fuzzing not only helps find new code, similar to state-of-the-art coverage-guided fuzzers [9, 36], but can also efficiently flip sanitizer checks and trigger bugs.

In this paper we present the following contributions:

- We demonstrate a generic way of finding interesting fuzzing targets by relying on existing compiler sanitizer passes.
- We demonstrate a dynamic approach to build a precise control-flow graph used to steer the input towards our targets.
- We implement ParmeSan, the first sanitizer-guided fuzzer using a two-stage directed fuzzing strategy to efficiently reach all the interesting targets.
- We evaluate ParmeSan, showing that our approach finds the same bugs as state-of-the-art coverage-guided and directed fuzzers in less time.

To foster further research, our ParmeSan prototype is open source and available at https://github.com/vusec/parmesan.

2 Background

2.1 Fuzzing strategy

In its most naive form blackbox fuzzing randomly generates inputs, hoping to trigger bugs (through crashes or other error conditions). The benefit of blackbox fuzzing is that it is easily compatible with any program.

On the other side of the spectrum we have whitebox fuzzing [6, 21], using heavyweight analysis, such as symbolic execution to generate inputs that triggers bugs, rather than blindly testing a large number of inputs. In practice, whitebox fuzzing suffers from scalability or compatibility issues (e.g., no support for symbolic execution in libraries/system calls) in real-world programs.

To date, the most scalable and practical approach to fuzzing has been greybox fuzzing, which provides a middle ground between blackbox and whitebox fuzzing. By using the same scalable approach as blackbox fuzzing, but with lightweight heuristics to better mutate the input, greybox techniques yield scalable and effective fuzzing in practice [5, 7, 17, 30].

The best known coverage-guided greybox fuzzer is American Fuzzy Lop (AFL) [47], which uses execution tracing information to mutate the input. Some fuzzers, such as Angora [9] and VU+zer [36], rely on dynamic data-flow analysis (DFA) to quickly generate inputs that trigger new branches in the program, with the goal of increasing code coverage. While coverage-guided fuzzing might be a good overall strategy, finding deep bugs might take a long time with this strategy. Directed fuzzers try to overcome this limitation by steering the fuzzing towards certain points in the target program.

2.2 Directed fuzzing

Directed fuzzing has been applied to steering fuzzing towards possible vulnerable locations in programs [7, 13, 18, 19, 41, 45]. The intuition is that by directing fuzzing towards certain interesting points in the program, the fuzzer can find specific bugs faster than coverage-guided fuzzers. Traditional directed fuzzing solutions make use of symbolic execution, which, as mentioned earlier, suffers from scalability and compatibility limitations.

AFLGo [4] introduces the notion of Directed Greybox Fuzzing (DGF), which brings the scalability of greybox fuzzing to directed fuzzing. There are two main problems with DGFs. The first problem is finding interesting targets. One possibility is to use specialized static analysis tools to find possible dangerous points in programs [13, 16]. These tools, however, are often specific to the bugs and programming languages used. Other approaches use auxiliary metadata to gather interesting targets. AFLGo, for example, suggests directing fuzzing towards changes made in the application code (based on git commit logs). While an interesting heuristic for incremental fuzzing, it does not answer the question when fuzzing an application for the first time or in scenarios without a well-structured commit log. The second problem is distance calculation to the interesting targets to guide the DGF. Static analysis might yield a sub-optimal view of the program. More concretely, the (interprocedural) CFG is either an overapproximation [8] or an underapproxi-
ponents and their interactions. There are three main com-
ponents: the target acquisition, the dynamic CFG and the fuzzer components. In this section, we briefly present a high-
level overview of each component and defer their design de-
tails to the following sections.

3.1 Target acquisition

The first component of our pipeline, target acquisition, col-
lects a number of interesting targets that we want our fuzzer
to reach. The set of targets is generated by the instrumen-
tation operated by the given sanitizer on the given program.
We use a simple static analysis strategy to compare the in-
strumented version of the program with the baseline and au-
tomatically locate the instrumentations placed by the san-
itizer all over the program. Next, target acquisition uses
pruning heuristics to weed out uninteresting instrumenta-
tions (e.g., “hot” paths less likely to contain bugs [44]) and
derive a smaller set of interesting targets for efficient fuzzing.
Section 4 details our target acquisition design.

3.2 Dynamic CFG

The second component of our pipeline, dynamic CFG, main-
tains a precise, input-aware CFG abstraction suitable for
“many-target directed fuzzing” during the execution of the
target program. We add edges to our CFG as we observe
them during the execution, and rely on DFA [1] to track de-
pendencies between the input and the CFG. As a result the
dynamic CFG component can track input-dependent CFG
changes and provide feedback to input mutation on which
input bytes may affect the CFG for a given input. Section 5
details our dynamic CFG design.

3.3 Fuzzer

The final component of our pipeline, the ParmeSan fuzzer,
takes an instrumented binary, the set of targets, an initial
distance calculation, and a set of seeds as input. Our fuzzing
strategy starts with input seeds to get an initial set of exe-
cuted basic blocks and the conditions covered by these ba-
sic blocks. It then tries to steer the execution towards tar-
gets from the target acquisition component using the pre-
cise distance information that is provided by the dynamic
CFG component. At each trial, the ParmeSan fuzzer priori-
tizes the solving of that condition from the list of the visited
conditions that results in the best distance to the target basic
blocks.

Since we already need DFA for CFG construction, we
can also use it to solve branch constraints. In ParmeSan,
this intuition is used not just to find new code to reach the
targets efficiently—similar to DFA-based coverage-guided
 fuzzers [9, 36]—but also to quickly flip the reached target
sanitizer checks and trigger bugs. The output of the fuzzer

2.3 Target selection with sanitizers

Modern compilers, such as GCC and Clang+LLVM ship
with a number of so-called sanitizers, that employ runtime
tests to detect possible bugs that cannot always be found
through static analysis. Sanitizers have been successfully
used for finding bugs [42] and have been used to improve
the bug-finding ability of fuzzers [38]. Typically these are
mainly deployed during testing, as the overhead can be sig-
nificant.

The sanitizer typically instruments the target program,
adding a number of checks for vulnerabilities such as buffer
overflows or use-after-free bugs (see Listing 1 for an exam-
ple of the instrumentation). If a violation occurs, the sanitizer
typically reports the error and aborts the program. ParmeSan
shows that sanitizers are useful not only to enhance a fuzzer’s
bug-finding capabilities, but also to improve the efficiency of
the fuzzing strategy to reduce the time-to-exposure (TTE) of
bugs.

2.4 CFG construction

Directed fuzzers take the distance to the targets into account
when selecting seeds to mutate. For example, AFLGo [4]
and HawkEye [8] use lightweight static instrumentation to
calculate the distance of a certain seed input to the specified
targets. This instrumentation relies on a static analysis phase
that determines the distance for each basic block to the se-
lected targets.

Many real-world applications, however, rely on indirect
calls for function handlers. A prime example are (web)
servers, where a number of different handlers are registered
based on the server configuration.

AFLGo [4] follows the former strategy, underapproximat-
ing the real CFG. HawkEye [8] follows the latter strategy,
overapproximating the real CFG. For this purpose, Hawkeye
uses points-to analysis to generate a CFG for indirect calls.
Context-sensitive and flow-sensitive analysis is too expen-
sive to scale to large programs. While complete, context-
sensitive analysis causes an indirect call to have many out-
going edges, possibly yielding execution paths that are not
possible for a given input. For example, if a configuration
file determines the function handler, the call may in prac-
tice only have one valid target site. We propose a dynamic
CFG construction approach augmented with dynamic data-
flow analysis (DFA) to address this problem.

3 Overview

Figure 1 presents a high-level overview of the ParmeSan
sanitizer-guided fuzzing pipeline, with the different com-
ponents and their interactions. There are three main com-
consists of generated *error inputs*. Section 5 details our fuzzing design.

### 4 Target acquisition

Our target acquisition component relies on off-the-shelf compiler sanitizers to find interesting targets to reach. The key idea is to direct the fuzzer towards triggering error conditions in the sanitizer and find real-world bugs in a directed fashion. By implementing the analysis in a generic way, we can use any existing or future sanitizer to collect possible interesting targets. Since our approach is entirely sanitizer-agnostic, we can easily retarget our fuzzing pipeline to a different class (or classes) of bugs depending on the sanitizer used.

#### 4.1 Finding instrumented points

Compiler frameworks, such as LLVM [25], transform the frontend code (written in languages such as C, Rust, etc.) to a machine-agnostic intermediate representation (IR). The analysis and transformation passes, such as sanitizers, generally work at the IR level. Suppose we take an application and transform it into LLVM IR. Existing sanitizer passes can then instrument the IR to add sanitization checks and enable runtime bug detection. For example, the snippet in Listing 1 has been augmented with UBSan [2] instrumentation to detect pointer overflows. The UBSan pass adds a conditional branch before loading a pointer (at %6). The added branch calls the error handling function `__ubsan_handle_pointer_overflow()` if the added conditional is met (i.e., an overflow occurs).

Sanitizers instrument programs in two different ways. Some instrumentations simply update internal data structures (e.g., shadow memory), while other instrumentations are used when the sanitizers detect the actual bug using a branch condition that either interacts with the internal sanitizer data structures (e.g., ASan’s out of bound access detection) or the immediate state of the program (e.g., Listing 1). Our goal is to direct fuzzing towards points where the sanitizer updates its internal data structure (i.e., interesting code paths) and the conditional branches that are introduced by the sanitizers which if solved mean that we have discovered a bug. We discuss how ParmeSan uses this intuition for effi-

Listing 1: LLVM IR without and with UBSan instrumentation to check for pointer overflows

```llvm
; ... Non-sanitized
%4 = load i8*, i8** %2, align 8
%5 = getelementptr inbounds i8, i8*, i64 %4
%6 = ptrtoint i8* %4 to i64
%7 = add i64 %6, %8
%9 = icmp uge i64 %7, %6
%10 = select i1 true, i1 %8, i1 %9
br i1 %10, label %12, label %11
<br label>:11: ; preds = %1
    call void @__ubsan_handle_pointer_overflow (...)
    br label %12
<br %17: ...SANITIZED...>
```

... Sanitized with UBSan

```llvm
%4 = load i8*, i8** %2, align 8
%5 = getelementptr inbounds i8, i8*, i64 %4
%6 = ptrtoint i8* %4 to i64
%7 = add i64 %6, %8
%9 = icmp uge i64 %7, %6
%10 = select i1 true, i1 %8, i1 %9
br i1 %10, label %12, label %11
<br label>:11: ; preds = %1
    call void @__ubsan_handle_pointer_overflow (...)
    br label %12
<br %17: ...>
```

Listing 1: LLVM IR without and with UBSan instrumentation to check for pointer overflows
cient fuzzing in Section 6.

Since there exist numerous different sanitizers, with new ones being added frequently, we want a sanitizer-agnostic analysis method to collect these targets. We do this by implementing a blackbox analysis of the IR difference (diff) of the target program compiled with and without the sanitizer. To include the instrumented basic blocks that do not include a conditional, we add all the predecessor basic blocks instrumented by the sanitizer. For instrumented basic blocks that include a conditional, we include both the instrumented basic block and the basic block with a taken conditional (i.e., often the sanitizer’s bug checking function). We found this a simple strategy to yield a generic and effective way to obtain targets that is compatible with all the existing (and future) LLVM sanitizers.

### 4.2 Sanitizer effectiveness

To verify that our approach of using sanitization instrumentation as interesting targets is sound, we instrumented a number of applications, and confirmed that the targeted sanitizer checks detect the actual bugs. In Table 1, we tested the effectiveness of three different sanitizers against a number of known vulnerabilities.

AddressSanitizer (ASan) [37] is able to discover buffer overflows and use-after-free bugs. UndefinedBehaviorSanitizer (UBSan) [2] is able to detect undefined behavior, such as using misaligned or null pointers, integer overflows, etc. The Type Sanitizer (TySan) [14] is able to detect type confusion when accessing C/C++ objects with a pointer of the wrong type.

Table 1 shows whether the sanitizer catches the bug and the number of basic blocks of the program not contained in a path to instrumented basic blocks. For example, if a deep basic block is considered a target (i.e., contains a target branch), all its predecessors have to be covered. However, non-target basic blocks that are not on a path to a target do not need to be covered, as our analysis estimates there are no bugs in those blocks. By calculating the number of basic blocks that we can disregard (non-target) in this way, we get a metric estimating how many basic blocks are irrelevant for triggering sanitizer errors, and are thus not necessary to be covered when fuzzing. This metric gives us an estimate of how sanitizer-guided fuzzing compares against traditional coverage-oriented fuzzing for different sanitizers.

In many cases, a significant part of the code coverage can be disregarded. For example in libxml2 using TySan, we can disregard 80% of the basic blocks and still find the bug. However, as seen in the pruning metric in Table 1, there is a major variance in how much of the application different sanitizers instrument. Some sanitizers, such as UBSan and TySan, are specialized in what they instrument, yielding a small set of targets. Other sanitizers, such as ASan, instrument so many basic blocks that, if we were to consider every instrumented point a target, we would essentially end up with coverage-guided fuzzing.

Thus, the challenge is to limit the number of acquired targets to consider, while still keeping the interesting targets that trigger actual bugs. To address this challenge, our solution is to adopt pruning heuristics to weed out targets part of the candidate target set. We experimented with a number of pruning heuristics and ultimately included only two simple but effective heuristic in our current ParmeSan prototype.

### 4.3 Profile-guided pruning

Our first heuristic to limiting the number of targets is to perform profile-guided target pruning. By applying a similar approach to ASAP [44], our strategy is to profile the target program and remove all the sanitizer checks on hot paths (i.e., reached by the profiling input). Since hot paths are unlikely to contain residual bugs that slipped into production [27, 44], this strategy can effectively prune the set of targets, while also preferring targets that are “deep”/hard-to-reach. While this pruning mechanisms might remove some valid targets, the authors of ASAP [44] note that (in the most conservative estimate) 80% of the bugs are still detected.

### 4.4 Complexity-based pruning

Our second heuristic to limiting the number of targets is to operate complexity-based pruning. Since sanitizers often add other instrumentation besides a simple branch, we score functions based on how many instructions are added/modified by the sanitizer (diff heuristic) and mark targets that score higher than others as more interesting. The intuition is that the more instructions are changed within a function by the sanitizer, the higher the complexity of the function and thus the chances of encountering the classes of bugs targeted by the sanitizer. We show this intuition on LAVA-M [15] using ASan. Using the this heuristic, our top 3 targets in base64 are in the functions lava_get(), lava_set(), and emit_bug_reporting_address(), of which the top 2 func-

<table>
<thead>
<tr>
<th>Prog</th>
<th>Bug</th>
<th>Type</th>
<th>Sanitizer (% non-target)</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64</td>
<td>lava-M</td>
<td>BO</td>
<td>✓</td>
</tr>
<tr>
<td>who</td>
<td>LAVA-M</td>
<td>BO</td>
<td>✓</td>
</tr>
<tr>
<td>uniq</td>
<td>LAVA-M</td>
<td>BO</td>
<td>✓</td>
</tr>
<tr>
<td>libxml2</td>
<td>LAVA-M</td>
<td>BO</td>
<td>✓</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>2014-0160</td>
<td>BO</td>
<td>✓</td>
</tr>
<tr>
<td>pcre2</td>
<td>-</td>
<td>UAF</td>
<td>✓</td>
</tr>
<tr>
<td>libxml2</td>
<td>memleak</td>
<td>TC</td>
<td>✓</td>
</tr>
<tr>
<td>libpng</td>
<td>oom</td>
<td>IO</td>
<td>✓</td>
</tr>
<tr>
<td>libarchive</td>
<td>-</td>
<td>BO</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Bugs detected and percentage of branches that can be disregarded (i.e., are not on the path to an instrumented basic block) compared to coverage-oriented fuzzing. UAF= use-after-free, BO=buffer overflow, TC=type confusion, IO=integer overflow.
tions are the functions in LAVA-M that trigger the injected bugs. The score is taken into consideration when selecting which targets to prune based on profiling. This allows our target acquisition component to be geared towards retaining targets in cold code.

5 Dynamic CFG

To make our sanitizer-guided fuzzing strategy effective, ParmeSan must be able to efficiently steer the execution towards code that is identified by the target acquisition step. To do this, ParmeSan needs a precise CFG to estimate the distance between any given basic block and the target. Building a precise CFG is the role of our dynamic CFG component. We first show how we dynamically improve the CFG’s precision during fuzzing (Section 5.1). Using the improved CFG, ParmeSan then needs to make use of a distance metric to decide which code paths to prioritize given how far an execution trace is from interesting code blocks that are instrumented by sanitizers (Section 5.2). To further improve the quality of ParmeSan’s distance metric, we augment our CFG with Dynamic (Data-)Flow Analysis (DFA) information to ensure certain interesting conditions are always satisfied by selecting the current input bytes (Section 5.3).

5.1 CFG construction

Prior directed fuzzers rely on a statically-generated CFGs for distance calculation. In directed fuzzing with many targets, statically-generated CFGs lead to imprecise results. For ParmeSan, we instead opt for a dynamically-generated CFG. In particular, we start with the CFG that is statically generated by LLVM, and then incrementally make it more precise by adding edges on the fly as the program executes during fuzzing. This addition of edges happens, for example, when we discover an indirect call which cannot be resolved statically during compile time.

To perform scalable distance calculations, we use the number of conditionals between a starting point and the target, as conditionals are the essence of what a fuzzer tries to solve. Compared to the full CFG, this strategy yields a compact Conditional Graph (CG)—a compacted CFG that only contains the conditionals. ParmeSan maintains both the CG and the CFG at runtime, but uses only the CG for distance calculations.

We repurpose the AFL edge coverage tracking strategy [47] for our compact CG design. After assigning a randomly generated identifier to each basic block, we initially collect them all from the CFG. Note that the number of nodes is static and will never change. The edges in the CFG, on the other hand, are dynamic, and we add them to the CG and CFG when we encounter edges that are not yet present. Specifically, for each edge that the execution takes, we log the edge identifier (a hash of the previous and current basic block identifiers) and if the edge is not yet in the CFG, we simply add it. When we add edges to the CFG, we only have to update a subset of the CG, adding only the missing edges for the neighboring conditionals of the new edge.

5.2 Distance metric

The distance metric helps the fuzzer decide which parts of the CFG it needs to explore next to get closer to the basic blocks of interest. Since distance calculation can quickly run into scalability issues, here we opt for a simple metric. We define the distance of a given branch condition $c$ to the branch conditions that lead to the interesting basic blocks as $d(c)$. To calculate $d(c)$, we follow a recursive approach in which the neighboring basic blocks of a target branch will have a weight of 1. The neighbors of the neighbors’ weights are then calculated using the harmonic mean (somewhat similar to the one used by AFLGo [4]). Implementationwise, the results in the calculation are propagated starting from the targets, keeping track of which edges have already been propagated. During implementation, we empirically tested a few distance metrics, and found the following to be both scalable and accurate.

Let $N(c)$ be the set of (yet unaccounted for) successors of $c$ with a path to at least one of the targets, then:

$$d(c) = \begin{cases} 0 & \text{if } c \in \text{Targets} \\ \infty & \text{if } N(c) = \emptyset \\ \left(\frac{\sum_{n \in N(c)} d(n)^{-1}}{|N(c)|}\right)^{-1} + 1 & \text{otherwise} \end{cases}$$

Given an execution trace for a given input, ParmeSan uses the distance metric to determine which of the branches it should try to flip (by modifying the input), steering the execution towards interesting basic blocks. While our evaluation (Section 8) shows that even such a simplistic distance metric works well, we expect that better scheduling might lead to better performance. We leave this problem as an open question for future work.

5.3 Augmenting CFG with DFA

Our dynamic CFG can further improve distance calculation by fixing the indirect call targets to a single target depending on the input. If we know both the sanitizer check that we want to reach and the input bytes that determine the target of an indirect call, we can fix the input bytes such that we know the target of the indirect call. This simple improvement can drastically impact the precision of our distance calculation. This optimization is mainly beneficial if the program has many indirect calls with many possible targets.
distance information provided by the dynamic CFG priority queue containing entries consisting of a conditional.

The main fuzzing loop repeatedly pops an entry from the priority queue containing entries consisting of a conditional, the distance, and the corresponding seed that uncovered that conditional. The queue is sorted based on a tuple consisting of (runs, distance), where runs is the number of times this entry has been popped from the queue and distance is the calculated distance of the conditional to our targets obtained by using our dynamic CFG.

In the fuzzing loop, ParmeSan pops the entry with the lowest priority from the queue. Using the number of runs as the first key when sorting ensures that the fuzzer does not get stuck on a single conditional with a low distance. This is an effective way to mimic coverage-guided, while giving priority to promising targets.

The fuzzer then mutates the selected seed, giving priority to input bytes that affect the conditional (as provided by DFA), with the goal of triggering new coverage. If the fuzzer generates an input that increases coverage, we add the input and its coverage to the list of candidate inputs that we will consider adding to the queue.

We do a DFA-instrumented run for each of these inputs to collect the taint information for the new basic blocks the input uncovers. While taint tracking is expensive, we only need to collect this when we find new code coverage. As finding new coverage is relatively rare, the amortized overhead of tracking is negligible (as discussed in Section 8). For every new conditional that the input covers, we add an entry consisting of the conditional, the distance, and the seed to the queue.

Finally, after the original seed has been mutated a number of times (set to 30) in the round we push it back onto the queue with an updated distance if the CFG has changed since the last run.

### 6.3 Efficient bug detection

We have discussed how ParmeSan uses compiler sanitizers to direct fuzzing towards interesting targets in the program. In other words, while sanitizers have been used for bug detection in existing fuzzing efforts (i.e., fuzzing a sanitized version of the program to improve bug detection beyond crash detection in the baseline) [38], ParmeSan uses compiler sanitizers for analysis purposes. Moreover, just like existing fuzzers, ParmeSan can fuzz the target program with or without sanitizers (with a trade-off between bug detection coverage and performance).

However, compared to existing fuzzers, ParmeSan can perform much more efficient sanitizer-based bug detection if desired. Since we know where the interesting sanitizer checks are, ParmeSan supports a simple but effective optimization (which we call lazysan). In particular, ParmeSan can enable sanitizer instrumentation on demand only when this is useful (i.e., when we reach the desired target checks) and run the uninstrumented version at full speed otherwise—similar in spirit to our DFA-enhanced input mutation strategy.
6.4 End-to-end workflow

The end-to-end fuzzing workflow consists of three phases, a short coverage-oriented exploration and tracing phase to get the CFG (only run for the input seeds), a directed exploration phase to reach the target basic blocks, and an exploitation phase which gradually starts when any of the specified targets are reached.

During the short initial tracing phase, ParmeSan collects traces and tries to build a CFG that is as accurate as possible. During the directed exploration phase, ParmeSan tries to solve conditionals to reach the desired targets. The exploitation phase starts whenever ParmeSan reaches a target. ParmeSan tries to exploit the points reached so far by means of targeted DFA-driven mutations and, when configured to do so, also switches to the sanitizer-instrumented version of the program on demand. Note that the directed exploration stage and exploitation stage are interleaved. ParmeSan only performs the exploitation strategy for inputs that have reached the target, while still continuing to do exploration to reach open targets.

7 Implementation

We implement the fuzzing component of ParmeSan on top of Angora [9], a state-of-the-art coverage-guided fuzzer written in Rust. The blackbox sanitizer analysis consists of a number of Python scripts and LLVM passes. The modifications required to Angora consist of about 2,500 lines of code. We also integrate AFLGo into the ParmeSan pipeline, allowing us to use AFLGo as a fuzzing component, rather than the ParmeSan fuzzer, based on Angora.

To implement our target acquisition component, we run the llvm-diff tool between the sanitizer-instrumented and the uninstrumented version of the target program. We analyze the resulting LLVM IR diff file and label all the conditionals added by the instrumentation as candidate targets. We implement our target set pruning strategy on top of ASAP [44], which already removes sanitizer checks in hot paths to improve sanitizer-instrumented program performance. We augment ASAP, letting it take into account the complexity-based pruning heuristics described in Section 4.4 when deciding which checks to remove.

We base the fuzzer and dynamic CFG components of ParmeSan on Angora [9]. Angora keeps a global queue, consisting of pairs of conditionals (i.e., branching compare points) and input seeds. In Angora, these queue entries are prioritized based on how hard a conditional is to solve (e.g., how many times it has been run).

We modify Angora to sort queue entries by distance to the targets generated by the target acquisition step and direct fuzzing towards them. Furthermore, we added a dynamic CFG component to Angora, to allow for CFG constraint collection, making it possible to narrowly calculate distances to our targets based on the obtained coverage and the conditional to be targeted.

Similar to Angora, we use DataFlowSanitizer (DFSan) [1], a production DFA framework integrated in the LLVM compiler framework. We use such information in a dedicated LLVM instrumentation pass that traces each indirect call and records the input bytes that determine (i.e., taint) the target of the indirect call site. Note that we only run the DFSan-instrumented version of our program (for CFG construction or fuzzing) and re-calculate target distances when we uncover a new edge, resulting in low overhead.

7.1 Limitations

Currently, ParmeSan relies on available LLVM IR for its target acquisition. In theory the techniques described in this paper can also be applied to binaries without the IR available. While the analysis currently relies on compiler sanitizer passes, however, for raw binaries the methods we present could be applied by replacing the compiler sanitizers with binary hardening [33, 48]. We also noted an issue with some sanitizers that only insert their modifications at linking time; doing the analysis on the actual binary would solve this issue.

The types of bugs found by ParmeSan are heavily reliant on the sanitizers used for target acquisition (as we show in Section 8.3). Some sanitizers, such as ASan, are capable of detecting a broad class of common bugs. We refer the reader to [42] for a more thorough analysis on using sanitizers in a security context for testing and production purposes.

8 Evaluation

In this section we evaluate ParmeSan on a number of real-world programs with known bugs. We compare how ParmeSan performs against other directed and coverage-guided greybox fuzzers. We also show how our dynamic CFG construction improves fuzzing for real-world programs with pervasive indirect calls. Some additional results are presented in Appendix A.

We run all our experiments on machines running Ubuntu 18.10 using AMD 7 Ryzen 2700X with 32 GB DDR4 RAM. While both ParmeSan and Angora are able to use multiple cores, we run all our experiments on only one core to be able to compare against prior work, unless noted otherwise. For each part of the evaluation, we specify which sanitizer we use for target acquisition and repeat the experiments 30 times with a timeout of 48 hours, unless otherwise noted. During the profiling-guided pruning phase in our target acquisition component, we always set the ASAP cost level to 0.01. This is the equivalent of adding instrumentation at a cost of 1% in performance. As noted by the ASAP authors [44], this strategy sufficiently covers bugs, while aggressively removing hot checks. Note that the target acquisition step is not
included in the total run time of our benchmarks, as it is part of the compilation process. In all our experiments, the time spent on analysis is linear to the original compilation time of the target program (as shown in Table 8).

### 8.1 ParmeSan vs. directed fuzzers

We first compare against state-of-the-art directed greybox fuzzers and show the availability of DFA information alone improves directed fuzzing significantly. We reproduce a number of benchmarks covered by AFLGo [4] and HawkEye [8], showing how ParmeSan fares in a traditional directed setting. Note that the source code for HawkEye is not available at the moment, and thus we compare against the results reported by the authors. While comparisons to results in papers is difficult due to variations in the test setup, since the baseline performance of AFLGo presented by the HawkEye authors [8] is similar to the one we obtained in our setup, we are hopeful that their performance numbers are also comparable to ours.

To show that sanitizer-guided fuzzing can efficiently find real-world bugs, we evaluate ParmeSan on the Google fuzzer-test-suite [22]. This dataset contains a number of known bugs, coverage benchmarks, and assertion checks for 23 real-world libraries. We show that ParmeSan is able to trigger the same bugs as coverage-oriented fuzzers in significantly less time. In this suite, we always use ASan for ParmeSan’s target acquisition step, as it is very powerful and detects some of the most common memory errors.

In all benchmarks, we use the seeds provided by the suite as the initial corpus. Since the dataset contains a number of hard-to-trigger bugs, we run the experiments with a timeout of 48 hours, to give the fuzzers a chance at reaching these bugs. For example, it takes Angora on average 47 hours to trigger the integer overflow in freetype2. Furthermore, the suite adds runtime sanitizers to each application to detect the bugs. We compile and run every program with the default parameters used in the suite.

Table 3 shows the mean time-to-exposure (TTE) of a number of bugs from the Google fuzzer-test-suite dataset. We emphasize that we evaluated the entire test suite, but for brevity left out 11 bugs that no fuzzer could find within 48 hours, as well as the openthread set with its 12 very easy to find bugs which did not have any outlying results (of course, we did include them in our geometric mean calculation to avoid skewing the results). The evaluation is split into two parts. The first part, **whole pipeline**, uses the whole ParmeSan pipeline with automatic target acquisition using ASan. We compare ParmeSan against baseline Angora (i.e., no targets) and sanitizer-guided AFLGo (i.e., provided with
the same targets as ParmeSan). We see that ParmeSan outperforms both AFLGo and Angora significantly, with a geometric speedup in TTE of 288% and 37% respectively.

In the second part, we manually target a number of known hard-to-reach sites. These benchmarks from the suite check whether fuzzers are able to cover hard-to-reach sites or trigger assertion errors. Since in these cases there is no bug to be found, using a sanitizer-guided approach makes little sense. Instead, we show the effect of making the fuzzer directed. As these targets have to be selected manually, we consider the comparison against Angora to be unfair and only include the results as an indication how much directed fuzzing can help in such scenarios.

Interestingly, Angora beats AFLGo in every benchmark on the whole suite. The main cause for this is that Angora has access to DFA information which allows it to cover new branches much more quickly than the AFL-based strategy used by AFLGo. Note that some of our results when comparing ParmeSan against Angora are not statistically significant (Mann-Whitney p-value ≥ 0.05). All of these are bugs that are either triggered in a short amount of time (and thus have a large variance in the measurements), or are memory leaks (for which the immediate cause is independent of the targets retrieved by our target acquisition component, as we discuss in the next section). On the libbash benchmark, ParmeSan performs worse than Angora. This happens due to the fact that the bug is often triggered at a point when a lot of new coverage is found in one go. Due to our lazySan optimization, ASan is not enabled when this new coverage is triggered, causing ParmeSan to detect the bug later when it actually tries to flip the branch that causes the sanitizer error. As Table 7 shows, ParmeSan without the lazySan optimization is faster at finding this particular bug. Note that the variance in this test case is very high, and, as such, the result is not statistically significant.

In Table 4, we present branch coverage at the time-of-exposure (TTE) for ParmeSan and 4 different state-of-the-art fuzzers: AFLGo [4], NEUZZ [40], QSYM [46], and Angora [9]. In this experiment, we run all the fuzzers with 1 instance, except QSYM which uses 2 AFL instances and one QSYM instance (as per the setup suggested by the authors) inside a Docker container that has been allocated one CPU. Note that we do not include the required preprocessing time for NEUZZ and ParmeSan in the results. For ParmeSan, the

<table>
<thead>
<tr>
<th>Prog</th>
<th>Type</th>
<th>Runs</th>
<th>Mean. TTE AFLGo (p)</th>
<th>Mean. TTE Angora (p)</th>
<th>ParmeSan</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Whole pipeline</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boringssl</td>
<td>UAF</td>
<td>30</td>
<td>2h32m 0.004</td>
<td>45m 0.005</td>
<td>25m</td>
<td>crypto/asn1/asn1_lib.c:459</td>
</tr>
<tr>
<td>c-ares</td>
<td>BO</td>
<td>30</td>
<td>5s 0.04</td>
<td>1s 0.12</td>
<td>1s</td>
<td>CVE-2016-5180</td>
</tr>
<tr>
<td>freetype2</td>
<td>IO</td>
<td>5</td>
<td>x</td>
<td>47h 0.018</td>
<td>43h</td>
<td>cf2_doflex</td>
</tr>
<tr>
<td>pcre2</td>
<td>UAF</td>
<td>30</td>
<td>25m 0.006</td>
<td>15m 0.003</td>
<td>8m</td>
<td>src/pcre2_match.c:5968</td>
</tr>
<tr>
<td>lcms</td>
<td>BO</td>
<td>30</td>
<td>6m 0.002</td>
<td>2m 0.006</td>
<td>41s</td>
<td>src/cmsintrip.c:642</td>
</tr>
<tr>
<td>libarchive</td>
<td>BO</td>
<td>30</td>
<td>1h12m 0.004</td>
<td>22m 0.001</td>
<td>13m</td>
<td>archive_read_support_format_warc.c:537</td>
</tr>
<tr>
<td>libssh</td>
<td>ML</td>
<td>30</td>
<td>3m10s 0.002</td>
<td>32s 0.008</td>
<td>50s</td>
<td>CVE-2015-8317</td>
</tr>
<tr>
<td>libxml2</td>
<td>BO</td>
<td>30</td>
<td>51m 0.007</td>
<td>20m 0.001</td>
<td>11m</td>
<td>CVE-2015-8317</td>
</tr>
<tr>
<td>libxml2</td>
<td>ML</td>
<td>30</td>
<td>30m 0.005</td>
<td>20m 0.001</td>
<td>17m</td>
<td>memleak.valid.c:952</td>
</tr>
<tr>
<td>openssl-1.0.1f</td>
<td>BO</td>
<td>30</td>
<td>50m 0.003</td>
<td>5m 0.04</td>
<td>3m4s 0.004</td>
<td>CVE-2014-0160. OpenSSL 10.0.1f</td>
</tr>
<tr>
<td>openssl-1.0.1f</td>
<td>ML</td>
<td>30</td>
<td>1m 0.012</td>
<td>40s 0.11</td>
<td>37s</td>
<td>crypto/mem.c:308</td>
</tr>
<tr>
<td>proj4</td>
<td>ML</td>
<td>30</td>
<td>7m30s 0.002</td>
<td>1m40s 0.03</td>
<td>1m26s</td>
<td></td>
</tr>
<tr>
<td>re2</td>
<td>BO</td>
<td>30</td>
<td>47m 0.002</td>
<td>21m 0.004</td>
<td>12m35s</td>
<td></td>
</tr>
<tr>
<td>woff2</td>
<td>BO</td>
<td>30</td>
<td>45m 0.004</td>
<td>15m 0.006</td>
<td>8m</td>
<td></td>
</tr>
<tr>
<td>Geomean ParmeSan benefit</td>
<td></td>
<td></td>
<td>288%</td>
<td>37%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Time-to-exposure on the Google fuzzer-test-suite. For the tests under manual target, there is no actual bug, here we manually target the site (i.e., no target acquisition phase). Statistically significant Mann-Whitney U test p-values (p < 0.05) are highlighted. x= not found, — = not available. In all cases, we use ASan for target acquisition. UAF=use-after-free, BO=buffer overflow, IO=integer overflow, ML=memory leak, AE=assertion error.
when the program terminates. Note that LSan does not mod-
jects at runtime and generates a summary of memory leaks
(see Table
LeakSanitizer (LSan) instead of ASan for target acquisition
peat the experiment on the memory leak bugs, but now using
direct the fuzzing towards calls that allocate memory. We re-
tual use of allocated memory, but ideally we would like to
use ASan for target acquisition, the fuzzing will be directed
analysis has a significant impact on the end result. Since we
memory-leak bugs. This is a first indication that our sanitizer
show that the sanitizer used determines the classes of bugs
8.3 Sanitizer impact
We now take a look at how the particular sanitizer used in our
analysis impacts the final results of the fuzzing pipeline. We
show that the sanitizer used determines the classes of bugs
ParmeSan can find, allowing us to focus fuzzing on specific
types of bugs.
Table 3, shows ParmeSan performs the worst on the
memory-leak bugs. This is a first indication that our sanitizer
analysis has a significant impact on the end result. Since we
use ASan for target acquisition, the fuzzing will be directed
to possible invalid uses of memory. This still covers the ac-
tual use of allocated memory, but ideally we would like to
direct the fuzzing towards calls that allocate memory. We re-
peat the experiment on the memory leak bugs, but now using
LeakSanitizer (LSan) instead of ASan for target acquisition
(see Table 5). LSan keeps track of allocated memory ob-
jects at runtime and generates a summary of memory leaks
when the program terminates. Note that LSan does not mod-
ify the IR, but rather intercepts library calls to functions such
as malloc, which happens at link time. Instead, we create a
custom LLVM pass that inserts dummy calls to the hooks of the
interpreted functions, yielding the same behavior as normal
LSan while still changing the IR at the relevant locations.
This is a process that can be easily automated in the future,
and is a limitation only of the current implementation. With
our custom LSan pass for target acquisition, the mean TTE
for the memory leak bugs in libssh, libxml, openssl, proj4
then changes significantly, yields a geometric
improvement of 32% compared to using ASan for target acquisition.
Like-
wise for the integer overflow in freetype2, we see that us-
ing the correct sanitizer which actually catches the bug (i.e.,
UBSan) for target acquisition improves the performance sig-
ificantly, finding the bug in 20 hours rather than 47 hours.
As shown in Table 5, there is a stark contrast between san-
itzers used for target acquisition. We run a number of ap-
lications with known bugs of a certain type, while using
three different sanitizers (ASan, UBSan, and TySan) to aut-
omatically find targets. Note that triggering the bugs requires
sanitizers also (as the bugs usually do not crash the program).
To elimiate the variance caused by overhead of each sani-
tizer, we always instrument the program with the same set of
targets used for target acquisition. We run a number of ap-
lications with known bugs of a certain type, while using
three different sanitizers (ASan, UBSan, and TySan) to aut-
omatically find targets. Note that triggering the bugs requires
sanitizers also (as the bugs usually do not crash the program).
To elimiate the variance caused by overhead of each sani-
tizer, we always instrument the program with the same set of
runtime sanitizer (ASan + LeakSan + UBSan, which is able
to detect all the selected bugs), regardless of the one used for
target acquisition.
As shown in Table 5, a sanitizer that detects the bug
will always allow ParmeSan to find the bug within the least
amount of time. Overall, we see that using the sanitizers that
covers the bug and instruments a minimum set of targets al-
ows ParmeSan to find bugs faster.
For example, CVE-2018-13785 is a hard-to-trigger inte-

<table>
<thead>
<tr>
<th>Prog</th>
<th>Type</th>
<th>Runs</th>
<th>AFLGo</th>
<th>NEUZZ</th>
<th>QSYM</th>
<th>Angora</th>
<th>ParmeSan</th>
</tr>
</thead>
<tbody>
<tr>
<td>boringssl</td>
<td>UAF</td>
<td>10</td>
<td>2281</td>
<td>2h32m</td>
<td>2520</td>
<td>1h20m</td>
<td>2670</td>
</tr>
<tr>
<td>c-ares</td>
<td>BO</td>
<td>10</td>
<td>202</td>
<td>5s</td>
<td>275</td>
<td>3s</td>
<td>280</td>
</tr>
<tr>
<td>freetype2</td>
<td>IO</td>
<td>5</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>pcre2</td>
<td>UAF</td>
<td>10</td>
<td>9023</td>
<td>25m</td>
<td>31220</td>
<td>16m</td>
<td>32430</td>
</tr>
<tr>
<td>lcms</td>
<td>BO</td>
<td>10</td>
<td>1079</td>
<td>6m</td>
<td>2876</td>
<td>1m50s</td>
<td>3231</td>
</tr>
<tr>
<td>libarchive</td>
<td>BO</td>
<td>10</td>
<td>4870</td>
<td>1h12m</td>
<td>5945</td>
<td>1h20m</td>
<td>X</td>
</tr>
<tr>
<td>libssh</td>
<td>ML</td>
<td>10</td>
<td>365</td>
<td>3m10s</td>
<td>419</td>
<td>43s</td>
<td>631</td>
</tr>
<tr>
<td>libxml2</td>
<td>BO</td>
<td>10</td>
<td>5780</td>
<td>51m</td>
<td>7576</td>
<td>25m</td>
<td>12789</td>
</tr>
<tr>
<td>libxml2</td>
<td>ML</td>
<td>10</td>
<td>5755</td>
<td>30m</td>
<td>10644</td>
<td>19m</td>
<td>11260</td>
</tr>
<tr>
<td>openssl-l.1.0.1</td>
<td>BO</td>
<td>10</td>
<td>550</td>
<td>50m</td>
<td>814</td>
<td>10m12s</td>
<td>853</td>
</tr>
<tr>
<td>openssl-l.1.0.1</td>
<td>ML</td>
<td>10</td>
<td>1250</td>
<td>1m</td>
<td>717</td>
<td>40s</td>
<td>4570</td>
</tr>
<tr>
<td>proj4</td>
<td>ML</td>
<td>10</td>
<td>82</td>
<td>7m30s</td>
<td>83</td>
<td>1m55s</td>
<td>86</td>
</tr>
<tr>
<td>re2</td>
<td>BO</td>
<td>10</td>
<td>5172</td>
<td>47m</td>
<td>5178</td>
<td>50m</td>
<td>7610</td>
</tr>
<tr>
<td>wolf2</td>
<td>BO</td>
<td>10</td>
<td>91</td>
<td>45m</td>
<td>94</td>
<td>31m20s</td>
<td>98</td>
</tr>
<tr>
<td>wolf2</td>
<td>OOM</td>
<td>10</td>
<td>50</td>
<td>2m</td>
<td>50</td>
<td>22s</td>
<td>53</td>
</tr>
</tbody>
</table>

Table 4: Average branch coverage and TTE at the time of exposure for ParmeSan and several other state-of-the-art fuzzers. Compared to other fuzzers, ParmeSan requires a significantly lower coverage (and shorter time) to expose bugs. AFLGo uses the targets obtained using the ParmeSan analysis stage. All fuzzers run with sanitizers enabled.
the actual bug faster than for ASan, triggering the bug in the input generation is steered towards the target containing fewer targets obtained by TySan is smaller than for ASan, instrument the location of the vulnerability. Since the number of targets obtained by TySan is smaller than for ASan, the input generation is steered towards the target containing the actual bug faster than for ASan, triggering the bug in less time. CVE-2011-1944 is an integer overflow in libxml2, which is easy to trigger. Here, again, we see that the less-eager-to-instrument sanitizer lets ParmeSan trigger the bug in the least amount of time.

For CVE-2014-0160 (HeartBleed), on the other hand, we see that the sanitizer chosen does not have as significant an impact on how fast the bug is triggered. This is mainly due to the fact that ASan gives us a large number of targets. Note, that while fuzzing, we found a number of other crashes not related to HeartBleed, due to other memory errors. However, for CVE-2015-8317 (out-of-bounds heap read on libxml2), we see a major improvement, even though we get a large set of targets.

The interesting insight we get from these experiments is that ParmeSan is able to target specific kinds of bugs based on the sanitizer used for target acquisition and can thus be used to fuzz applications more effectively. For example, the use-after-free bug in pcre2 might manifest itself as a type confusion bug. Using Tysan for target acquisition, we are able to trigger the bug 20% faster. We have focused our analysis on a small number of common off-the-shelf sanitizers. For a more comprehensive overview of different sanitizers and behavior, we would like to point to the work of Song & al. [42].

### 8.4 New bugs

We apply ParmeSan to finding new bugs and compare the results with a number of state-of-the-art fuzzers using a selection of libraries. We include a random sampling of applications from OSS-Fuzz [39] and three target applications (jhead, pbc, protobuf-c) that were evaluated in recent work in fuzzing [3, 12, 32] in which we were able to uncover new bugs. We setup ParmeSan to fuzz the most recent commits on the master branch of the applications from the OSS-Fuzz sample. In total, ParmeSan was able to uncover 736 crashes, of which we determined 47 to be unique based on the call stack. Of these crashes 37 were found in the (somewhat) outdated pbc library, while 10 of them were found in up-to-date well-fuzzed libraries. The bugs found in the OSS-Fuzz applications, jhead, and protobuf-c have all been been triaged and resolved.

We emphasize that our analysis here (and in general evaluating a fuzzer on the number of new bugs found) on selected targets only serves as a case study and is not useful to assess the overall fuzzing performance—given that the selection of the targets and their particular versions can heavily influence the results. We refer the reader to the previous subsections for experiments detailing ParmeSan’s overall fuzzing performance.

Overall, our results show that ParmeSan outperforms other state-of-the-art directed greybox fuzzers by adding DFA information and dynamic control-flow graph construction. We have shown that directing fuzzing towards targets achieved by a sanitizer-guided analysis is an effective bug-finding strategy, allowing us to outperform state-of-the-art coverage-

---

<table>
<thead>
<tr>
<th>Bug</th>
<th>Type</th>
<th>Sanitizer</th>
<th>Targets</th>
<th>Covered</th>
<th>µTTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2014-0160</td>
<td>BO</td>
<td>ASan</td>
<td>533</td>
<td>✓</td>
<td>5m</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UBSan</td>
<td>120</td>
<td>X</td>
<td>6m</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TySan</td>
<td>5</td>
<td>X</td>
<td>6m</td>
</tr>
<tr>
<td>CVE-2015-8317</td>
<td>BO</td>
<td>ASan</td>
<td>352</td>
<td>✓</td>
<td>10m</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UBSan</td>
<td>75</td>
<td>X</td>
<td>50m</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TySan</td>
<td>30</td>
<td>X</td>
<td>50m</td>
</tr>
<tr>
<td>pcre2</td>
<td>UAF</td>
<td>ASan</td>
<td>122</td>
<td>✓</td>
<td>10m</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UBSan</td>
<td>52</td>
<td>X</td>
<td>20m</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TySan</td>
<td>12</td>
<td>✓</td>
<td>8m</td>
</tr>
<tr>
<td>freetype2</td>
<td>IO</td>
<td>ASan</td>
<td>437</td>
<td>✓</td>
<td>47h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UBSan</td>
<td>48</td>
<td>✓</td>
<td>20h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TySan</td>
<td>71</td>
<td>X</td>
<td>&gt;48h</td>
</tr>
<tr>
<td>CVE-2011-1944</td>
<td>IO</td>
<td>ASan</td>
<td>230</td>
<td>✓</td>
<td>30s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UBSan</td>
<td>125</td>
<td>✓</td>
<td>20s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TySan</td>
<td>8</td>
<td>X</td>
<td>50s</td>
</tr>
<tr>
<td>CVE-2018-13785</td>
<td>IO</td>
<td>ASan</td>
<td>450</td>
<td>X</td>
<td>11h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UBSan</td>
<td>45</td>
<td>✓</td>
<td>32m</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TySan</td>
<td>31</td>
<td>✓</td>
<td>5h</td>
</tr>
<tr>
<td>libssh</td>
<td>ML</td>
<td>ASan</td>
<td>590</td>
<td>X</td>
<td>31s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UBSan</td>
<td>57</td>
<td>X</td>
<td>33s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TySan</td>
<td>13</td>
<td>X</td>
<td>35s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSan</td>
<td>104</td>
<td>✓</td>
<td>25s</td>
</tr>
<tr>
<td>libxml</td>
<td>ML</td>
<td>ASan</td>
<td>352</td>
<td>X</td>
<td>15m</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UBSan</td>
<td>75</td>
<td>X</td>
<td>22m</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TySan</td>
<td>30</td>
<td>X</td>
<td>25m</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSan</td>
<td>191</td>
<td>✓</td>
<td>12m</td>
</tr>
<tr>
<td>openssl</td>
<td>ML</td>
<td>ASan</td>
<td>533</td>
<td>X</td>
<td>40s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UBSan</td>
<td>120</td>
<td>X</td>
<td>50s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TySan</td>
<td>5</td>
<td>X</td>
<td>43s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSan</td>
<td>191</td>
<td>✓</td>
<td>32s</td>
</tr>
<tr>
<td>proj4</td>
<td>ML</td>
<td>ASan</td>
<td>729</td>
<td>X</td>
<td>1m30s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UBSan</td>
<td>170</td>
<td>X</td>
<td>1m55s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TySan</td>
<td>373</td>
<td>✓</td>
<td>2m10s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSan</td>
<td>43</td>
<td>✓</td>
<td>57s</td>
</tr>
</tbody>
</table>

Table 5: Bugs found by ParmeSan using different sanitizers in the analysis stage. ✓ in targets, bug found; X not in targets, bug found; For the memory leak (ML) bugs we also show the performance of LeakSanitizer.
However, ParmeSan relies on such information to augment flow and data-flow analysis to augment the fuzzing process. ParmeSan implements a similar gradient-guided optimization as Angora. The behavior of a target application and uses this information to implement neural networks to approximate the discrete branching behavior of a target application and uses this information to implement a similar gradient-guided optimization as Angora.

NEUZZ [40] uses neural networks to approximate the discrete branching behavior of a target application and uses this information to implement a similar gradient-guided optimization as Angora. Similarly to Matryoshka [10], ParmeSan relies on control-flow and data-flow analysis to augment the fuzzing process. However, ParmeSan relies on such information to augment the CFG and fixing indirect calls, rather than using it to solve constraints.

**Directed Greybox Fuzzing** Böhme & al. introduce directed greybox fuzzing [4] with AFLGo. AFLGo takes a set of predetermined targets and tries to guide the fuzzing process in that direction. Unlike ParmeSan, AFLGo cannot operate as a drop-in replacement for coverage-guided fuzzing, as it includes no generic target acquisition analysis. Hawk-eye [8] improves upon the ideas in AFLGo by supporting indirect calls using static alias analysis. While Hawk-eye supports reaching targets via indirect calls, unlike ParmeSan’s dynamic CFG distance calculation, the static call-target analysis incurs overapproximations and does not take the input seed into account for distance calculation.

Driller [43] introduces hybrid fuzzing. By only using symbolic execution selectively for a smaller compartment of the total program, it is able to avoid path explosion common to prior symbolic execution approaches, and is thus able to scale to larger programs. KATCH utilizes static analysis and symbolic execution to generate inputs for increasing patch test coverage [29]. QSYM [46] introduces a new symbolic execution engine tailored to hybrid fuzzing, which is able to scale to larger programs than previous attempts at symbolic execution. TaintScope [45] uses tainting and symbolic execution to avoid the target program exiting an early stage due to invalid checksums in the input. A similar approach is taken by T-Fuzz [34], which transforms the target program by removing hard-to-solve checks to more easily reach possible bugs in the program. After a possible bug is found, T-Fuzz tries to reconstruct the input with symbolic execution such that the input passes the checks and triggers the deep bug.

Another use case for sanitizers in fuzzing that builds on similar ideas is the concurrent work presented by Chen et al. in SAVIOR [11], which suggests using the UBSan sanitizer to improve hybrid fuzzing. It solves constraints for UBsan checks to direct the fuzzing process towards actual bugs, avoiding costly concolic execution for many branches that are less prone to bugs. Note that this approach is not directly applicable to sanitizers, such as ASAN, that use internal datastructures (e.g., shadow memory). In contrast, ParmeSan’s generic dynamic taint tracking strategy makes it sanitizer-agnostic. This allows ParmeSan to use all available LLVM sanitizers for more fine-grained targeting of bug classes as shown in Section 8.3.

In a similar manner to ParmeSan, Hercules [35] uses dynamic CFG reconstruction techniques to reach bugs. While Hercules focuses on bug reproducibility (i.e., generating a crashing input given a target application and a crash report), ParmeSan focuses on finding bugs without the knowledge that a certain crash exists (i.e., generating a crash given a target application). Hercules augments the CFG with indirect calls and tainting information to satisfy conditions for reach-

<table>
<thead>
<tr>
<th>Prog</th>
<th>Version</th>
<th>Bugs</th>
<th>NEUZZ 1h</th>
<th>QSYM 1h</th>
<th>Angora 1h</th>
<th>ParmeSan 1h</th>
</tr>
</thead>
<tbody>
<tr>
<td>curl</td>
<td>54c622a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>json-c</td>
<td>d0d0490</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>libguestfs</td>
<td>804f40f3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>libxml2</td>
<td>10bf40</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>libcpp</td>
<td>cd27fd0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>6e4f1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ffinp</td>
<td>9d92403</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>harfbuzz</td>
<td>b21c5ef</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libpng</td>
<td>3301f7a1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 6:** New bugs found within 1h and 24h by ParmeSan and other state-of-the-art fuzzers. The version is denoted by either a version number or a commit id. In total ParmeSan found 47 new bugs.

oriented fuzzers as well. We have seen that ParmeSan can be between 37% to 876% faster at triggering bugs than other state-of-the-art fuzzer. In two cases, ParmeSan could find bugs that none of the other fuzzers could find.

## 9 Related work

In the software engineering community, search-based test data generation has been common for a number of years [24, 30, 31]. In a security context this approach is known as fuzzing.

**Greybox Fuzzing** Greybox fuzzing has been successfully applied to fuzzing a large number of programs [17, 47]. FairFuzz [26] augments AFL to prioritize seeds that exercise uncommon branches to improve branch coverage. Steelix [28] uses instrumentation to record comparison progress, allowing it to solve so-called “magic bytes” that need to be fixed not to quit the program at an early stage.

VUzzer [36] first suggested using dynamic data-flow analysis (DFA) in a greybox fuzzing strategy, allowing the input mutation to focus on the bytes that affect branches. ParmeSan shows DFA can also be used to accurately augment the control-flow graph for direct fuzzing purposes. REDQUEEN uses a lightweight input-to-state correspondence mechanisms as an alternative to data-flow analysis [3]. Angora [9] uses a gradient descent-based strategy to solve branch constraints in an efficient manner. NEUZZ [40] uses neural networks to approximate the discrete branching behavior of a target application and uses this information to implement a similar gradient-guided optimization as Angora.

Similarly to Matryoshka [10], ParmeSan relies on control-flow and data-flow analysis to augment the fuzzing process. However, ParmeSan relies on such information to augment the CFG and fixing indirect calls, rather than using it to solve constraints.
ing a target crash site. ParmeSan uses similar information, but instead uses it to improve distance calculations with better estimation of indirect call targets, given the input bytes that the fuzzer is mutating.

10 Conclusion

We presented ParmeSan, a sanitizer-guided greybox fuzzing pipeline. ParmeSan leverages off-the-shelf sanitizers, not only for detecting vulnerabilities as commonly used by prior fuzzers, but to actively guides the fuzzing process towards triggering the sanitizer checks. We identified a number of challenges in sanitizer-guided fuzzing, and discussed how ParmeSan addresses them. ParmeSan shows that off-the-shelf sanitizers are useful not only for bug detection, but for finding interesting fuzzing targets that match real-world bugs. ParmeSan trivially retargets the fuzzing strategy to different classes of bugs by switching to a different sanitizer, all in an automated and blackbox fashion. Our experimental results show that ParmeSan finds many classes of bugs with significantly lower time-to-exposure (TTE) than state-of-the-art fuzzers. ParmeSan is 37% faster than existing state-of-the-art coverage-based fuzzers (Angora) and 288% faster than directed fuzzers (AFLGo) when covering the same set of bugs. Techniques used by ParmeSan, such as taint-enhanced input mutation and dynamic CFG construction, can further benefit other fuzzers. To foster further research and encourage reproducibility, we will open-source ParmeSan upon acceptance of the paper.

11 Acknowledgments

We thank our shepherd, Aurélien Francillon, and the anonymous reviewers for their feedback. This work was supported by the EU’s Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct), by the Netherlands Organisation for Scientic Research through grants 639.023.309 VICI “Dowsing” and 639.021.753 VENI “PantaRhei”, by the United States Office of Naval Research under contract N00014-17-1-2782, and by Cisco Systems, Inc. through grant #1138109. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of any of the sponsors or any of their affiliates.

References


In this appendix, we include some additional results of our evaluation of different components of ParmeSan, as well as an evaluation of our target pruning strategy.

### A.1 Impact of different components

In Table 7, we present the results on the Google fuzzertest-suite, where we individually disable each of the three core components: lazy sanitizer optimization (lazysan), target pruning, and the dynamic CFG dyncfg. Overall, our results show that each component has a significant impact on fuzzing performance. Note that the lazysan optimization requires the dyncfg component.

When disabling the lazysan component, we see a degradation in TTE in almost every single case. The outliers are the bugs in libssh and the memory leak in openssl, where the performance improves when disabling lazysan. As discussed previously, this degradation in performance is due to the fact that the sanitizer is disabled when triggering the bug. Note that ParmeSan will still catch the bug, but triggering the sanitizer might be delayed until the exploitation phase.

Overall, we see that the different individual components each contribute significantly to the total performance of ParmeSan. For example, disabling the lazysan optimization, increases the TTE by 25%. Likewise, our target pruning accounts for 28% of the improvement. Without target pruning,
the behavior of ParmeSan becomes similar to baseline Angora, effectively emulating pure coverage-guided fuzzing.

By disabling the dyncfg component, we see an increase of 34% in TTE. Note that by disabling this component, we also effectively disable the lazysan component, as it relies on the control-flow information available by the dyncfg component. We further evaluate the added benefit of the dyncfg component in Section A.1.1.

### A.1.1 Dynamic CFG

Since ParmeSan uses a dynamic CFG to get a better estimate of the distance to the targets, we also want to show that the more accurate CFG actually improves the fuzzing process, rather than adding more overhead. The existing benchmarks—mostly C libraries—rarely contain a lot of indirect calls. However, in many applications (e.g., servers), indirect calls are common. We show the effect of dynamic CFG construction on three different experiments.

A.1.2 Comparison against SAVIOR

For the sake of completeness, we include Table 10, which shows how ParmeSan compares against Angora and SAVI-
Table 9: Time-to-exposure of bugs in programs where a number of direct calls have been "demoted". Apache httpd, cxxfilt, and boringssl have not been modified, as they already contain indirect calls. Statistically significant values ($p < 0.05$) are highlighted.

<table>
<thead>
<tr>
<th>Prog</th>
<th>Targets (pre-prune)</th>
<th>Targets (post-prune)</th>
<th>Bugs</th>
<th>Bugs Found</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1m</td>
<td>1h</td>
<td>24h</td>
<td></td>
</tr>
<tr>
<td>base64</td>
<td>1950</td>
<td>212</td>
<td>44</td>
<td>48</td>
</tr>
<tr>
<td>md5sum</td>
<td>1639</td>
<td>101</td>
<td>57</td>
<td>31</td>
</tr>
<tr>
<td>uniq</td>
<td>1832</td>
<td>193</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>who</td>
<td>2120</td>
<td>385</td>
<td>2136</td>
<td>1544</td>
</tr>
</tbody>
</table>

Table 10: Comparison of Angora, SAVIOR, and ParmeSan on LAVA-M. Mean number of LAVA-M bugs found over 10 24-hour runs using 3 parallel instances. We include results for ParmeSan for target acquisition using ASan, as well as explicitly targeting lava_get() (replicating the setup described in [11]).

Table 11: Analysis target pruning statistics and number of bugs found within 1 minute and within 24 hours. Some of the LAVA-M programs contain more bugs than specified in the dataset.