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 also 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 VUzzer [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 underapproximation [4] of the real one, leading to suboptimal fuzzing.

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 checks 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 significant.

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 example 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 selected 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, underapproximating 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 expensive to scale to large programs. While complete, context-insensitive analysis causes an indirect call to have many outgoing 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 practice 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 components and their interactions. There are three main components: 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 details to the following sections.

3.1 Target acquisition

The first component of our pipeline, *target acquisition*, collects a number of interesting targets that we want our fuzzer to reach. The set of targets is generated by the instrumentation operated by the given sanitizer on the given program. We use a simple static analysis strategy to compare the instrumented version of the program with the baseline and automatically locate the instrumentations placed by the sanitizer all over the program. Next, target acquisition uses pruning heuristics to weed out uninteresting instrumentations (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*, maintains 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 dependencies 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 executed basic blocks and the conditions covered by these basic blocks. It then tries to steer the execution towards targets from the target acquisition component using the precise distance information that is provided by the dynamic CFG component. At each trial, the ParmeSan fuzzer prioritizes 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

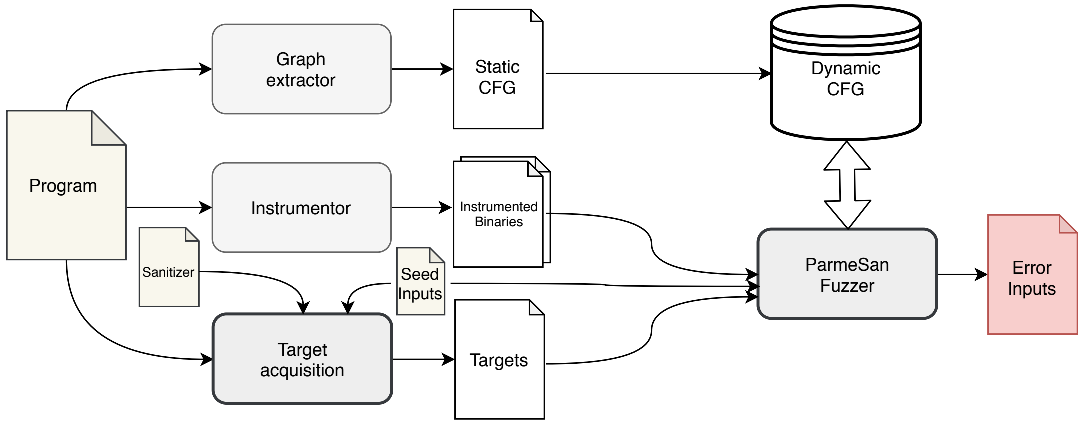

Figure 1: An overview of the ParmeSan fuzzing pipeline. The *target acquisition* step automatically obtains fuzzing targets. These targets are then fed to the ParmeSan fuzzer, which directs the inputs towards the targets by using the continuously updated *dynamic CFG*. The inputs to the pipeline consist of a target *program*, a *sanitizer*, and *seed inputs*.

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

```
;... Non-sanitized
%4 = load i8*, i8** %2, align 8
%5 = getelementptr inbounds i8, i8* %4, i64 1
%6 = load i8, i8* %5, align 1
;..
```

⇓

```
; ... Sanitized with UBSan
%4 = load i8*, i8** %2, align 8
%5 = getelementptr inbounds i8, i8* %4, i64 1
%6 = ptrtoint i8* %4 to i64
%7 = add i64 %6,
%8 = icmp uge i64 %7, %6
%9 = icmp ult i64 %7, %6
%10 = select i1 true, i1 %8, i1 %9
br i1 %10, label %12, label %11

; <label>:11:              ; preds = %1
call void @__ubsan_handle_pointer_overflow    (...)
br label %12

; ...
%17 = load i8, i8* %5, align 1
```

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

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 efficient 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

Prog	Bug	Type	Sanitizer (% non-target)					
			ASan		UBSan		TySan	
base64	LAVA-M	BO	✓	(5%)	✗	—	✗	—
who	LAVA-M	BO	✓	(9%)	✗	—	✗	—
uniq	LAVA-M	BO	✓	(15%)	✗	—	✗	—
md5sum	LAVA-M	BO	✓	(12%)	✗	—	✗	—
OpenSSL	2014-0160	BO	✓	(8%)	✗	—	✗	—
pcre2	-	UAF	✓	(7%)	✗	—	✗	—
libxml2	memleak	TC	✗	—	✗	—	✓	(80%)
libpng	oom	IO	✗	—	✓	(40%)	✗	—
libarchive	-	BO	✓	(17%)	✗	—	✗	—

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

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 functions 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 CFG and CG 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:

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

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.

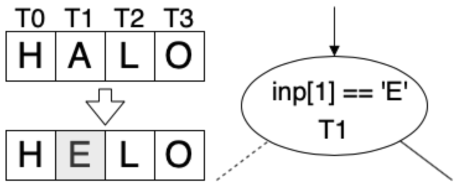

Figure 2: Example of DFA mutation. The taint label ($T1$) is recorded at a newly uncovered conditional, allowing the fuzzer to learn that the value should be either fixed to `E` or mutated further.

6 Sanitizer-guided fuzzer

In this section, we discuss how ParmeSan uses the targets obtained by the *target acquisition* component along with the distance information provided by the *dynamic CFG* component to direct the fuzzing process towards the desired targets and trigger bugs.

6.1 DFA for fuzzing

Existing directed greybox fuzzers [4, 8] show that directing the input based on simple distance metrics works well and is scalable. At the same time, existing DFA-based coverage-guided fuzzers [9, 36] show that adding DFA allows the fuzzer's input mutation to prioritize and solve tainted branch constraints in significantly fewer executions. When the fuzzer finds new coverage, the DFA-instrumented version of the program tracks the input byte offsets that affect the newly-found branches, such that the fuzzer can focus on mutating those offsets (see Figure 2). Since we already use DFA for augmenting the CFG, we also leverage the same analysis to implement coverage-guided-style DFA-enhanced input mutation but applied to (many-target) directed fuzzing. This allows us to focus the mutation on input bytes that affect conditionals, which will ultimately lead to our desired targets. Moreover, once we reach the desired target conditionals, we use DFA again to prioritize fuzzing of branch constraints, allowing us to trigger the bugs more efficiently.

Interestingly, we do not need a specialized mutation strategy to quickly flip sanitizer checks. Since we specifically target sanitizer-instrumented conditionals, we can simply use the same DFA-enhanced input mutation we used to reach the targets and get fast bug triggering "for free" as a by-product. Tainted sanitizer checks will automatically be prioritized, since tainted checks are preferred by DFA-enhanced input mutation and sanitizer checks are prioritized by our directed fuzzing strategy.

6.2 Input prioritization

The main fuzzing loop repeatedly pops an entry from the priority queue containing entries consisting of a *conditional* 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.

CVE	Fuzzer	Runs	p-val	Mean TTE
OpenSSL				
2014-0160	ParmeSan	30		5m10s
	HawkEye	—		—
	AFLGo	30	0.006	20m15s
Binutils				
2016-4487 2016-4488	ParmeSan	30		35s
	HawkEye	20		2m57s
	AFLGo	30	0.005	6m20s
2016-4489	ParmeSan	30		1m5s
	HawkEye	20		3m26s
	AFLGo	30	0.03	2m54s
2016-4490	ParmeSan	30		55s
	HawkEye	20		1m43s
	AFLGo	30	0.01	1m24s
2016-4491	ParmeSan	10		1h10m
	HawkEye	9		5h12m
	AFLGo	5	0.003	6h21m
2016-4492 2016-4493	ParmeSan	30		2m10s
	HawkEye	20		7m57s
	AFLGo	20	0.003	8m40s
2016-6131	ParmeSan	10		1h10m
	HawkEye	9		4h49m
	AFLGo	5	0.04	5h50m

Table 2: Reproduction of earlier results in crash reproduction in greybox fuzzers. We manually select the target and show the mean time-to-exposure.

In Table 2, we present a comparison of ParmeSan, AFLGo, and HawkEye on crash reproduction of known bugs in OpenSSL and Binutils. We manually target the point in the code that causes the crash, and let the fuzzers generate inputs to reproduce the crash (i.e., ParmeSan skips its target acquisition step). We use the same input seeds as presented in [8], consisting of a single file with a single newline character. As shown in the table, ParmeSan outperforms both HawkEye and AFLGo in reproducing these bugs in all cases. For most, ParmeSan is more than twice as fast, while in the worst case (CVE-2016-4490), it is still more than 30% faster at reproducing the bug than AFLGo. Adding DFA information allows ParmeSan to focus on solving conditionals, both on the way to the target and of the target itself—leading to a more targeted mutation strategy (fewer executions needed), allowing for faster crash reproduction. We conclude that ParmeSan significantly improves the state-of-the-art time-to-exposure (TTE) of bugs even for traditional directed fuzzing.

8.2 Coverage-guided fuzzers

We now show that our fuzzing strategy finds (many) bugs faster than state-of-the-art coverage-guided fuzzers. We specifically compare against Angora, which we found to be the fastest open-source competitor on the dataset considered, faster for instance than QSYM [46]. Note that if we target all the conditionals in the program, the behavior of ParmeSan is very similar to Angora. Comparing against Angora gives us a good picture of the effectiveness of targeting points obtained from our sanitizer-based analysis stage.

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 geomean 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

Prog	Type	Runs	Mean. TTE AFLGo (p)		Angora (p)		ParmeSan	Comment
Whole pipeline								
boringssl	UAF	30	2h32m	**0.004**	45m	**0.005**	25m	crypto/asn1/asn1_lib.c:459
c-ares	BO	30	5s	**0.04**	1s	0.12	1s	CVE-2016-5180
freetype2	IO	5	✗	—	47h	**0.018**	43h	cf2_doFlex.
pcre2	UAF	30	25m	**0.006**	15m	**0.003**	8m	src/pcre2_match.c:5968
lcms	BO	30	6m	**0.002**	2m	**0.006**	41s	src/cmsintrp.c:642
libarchive	BO	30	1h12m	**0.004**	22m	**0.001**	13m	archive_read_support_format_warc.c:537
libssh	ML	30	3m10s	**0.002**	32s	0.008	50s	
libxml2	BO	30	51m	**0.007**	20m	**0.001**	11m	CVE-2015-8317
libxml2	ML	30	30m	**0.005**	20m	**0.001**	17m	memleak. valid.c:952
openssl-1.0.1f	BO	30	50m	**0.003**	5m	**0.04**	3m4s	CVE-2014-0160. OpenSSL 10.0.1f
openssl-1.0.1f	ML	30	1m	**0.012**	40s	0.11	37s	crypto/mem.c:308
proj4	ML	30	7m30s	**0.002**	1m40s	**0.03**	1m26s	
re2	BO	30	47m	**0.002**	21m	**0.004**	12m35s	
woff2	BO	30	45m	**0.004**	15m	**0.006**	8m	
Geomean ParmeSan benefit			**288%**		**37%**			
Manual targeting								
libjpeg-turbo	⋆	30	1h8m	**0.003**	(45m)	**0.000**	10m	jdmarker.c:659
libpng	⋆	30	2m	**0.003**	(30s)	**0.002**	20s	pngread.c:738
libpng	⋆	30	2m	**0.005**	(42s)	**0.003**	34s	pngrutil.c:3182
freetype2	⋆	30	2s	0.21	(1s)	0.83	1s	ttgload.c:1710
guetzli	AE	30	45m	**0.000**	(10m)	**0.005**	5m	
harfbuzz	AE	30	5h	**0.000**	(2h20m)	**0.005**	1h10m	
json	AE	30	7m	**0.004**	(3m)	**0.005**	1m	
openssl-1.0.2d	AE	30	1m10s	**0.001**	(15s)	**0.04**	10s	CVE-2015-3193
Geomean ParmeSan benefit			**422%**		**90%**			

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. ✗= 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

the same targets as ParmeSan). We see that ParmeSan outperforms both AFLGo and Angora significantly, with a geomean 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 `libssh` 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

Prog	Type	Runs	AFLGo		NEUZZ		QSYM		Angora		ParmeSan	
boringssl	UAF	10	2281	2h32m	2520	1h20m	2670	3h20m	2510	45m	1850	25m
c-ares	BO	10	202	5s	275	3s	280	20s	270	1s	200	1s
freetype2	IO	5	✗	✗	✗	✗	✗	✗	57330	47h	49320	43h
pcre2	UAF	10	9023	25m	31220	16m	32430	1h20m	30111	15m	8761	8m
lcms	BO	10	1079	6m	2876	1m50s	3231	7m	2890	2m	540	41s
libarchive	BO	10	4870	1h12m	5945	1h20m	✗	✗	6208	22m	4123	13m
libssh	ML	10	365	3m10s	419	43s	631	2m32s	341	32s	123	50s
libxml2	BO	10	5780	51m	7576	25m	12789	2h5m	5071	20m	2701	11m
libxml2	ML	10	5755	30m	10644	19m	11260	1h10m	10580	20m	2554	17m
openssl-1.0.1f	BO	10	550	50m	814	10m12s	853	5h25m	793	5m	543	3m4s
openssl-1.0.1f	ML	10	1250	1m	717	40s	4570	23m	720	40s	709	37s
proj4	ML	10	82	7m30s	83	1m55s	86	10m5s	83	1m40s	80	1m26s
re2	BO	10	5172	47m	5178	50m	7610	2h	4073	21m	3267	12m35s
woff2	BO	10	91	45m	94	31m20s	98	41m	90	15m	83	8m
woff2	OOM	10	50	2m	50	22s	53	1m45s	50	20s	49	12s
Geomean diff			**+16%**	**+288%**	**+40%**	**+81%**	**+95%**	**+867%**	**+33%**	**+37%**		

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.

preprocessing time is in the order of the normal compilation time (as seen in Table 8). Every benchmark in the suite is run with the sanitizers enabled (as per the test suite).

In every single case except one, our results show that ParmeSan requires significantly less coverage to trigger the bug compared to the other state-of-the-art fuzzers. Likewise, AFLGo, which uses the targets obtained by the ParmeSan pipeline, also fares well in this regard, requiring slightly more coverage than ParmeSan to trigger the bugs. These results suggest that directing the fuzzing process towards sanitizer instrumentation reduces the coverage required to trigger bugs between 14 and 51%.

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 actual use of allocated memory, but ideally we would like to direct the fuzzing towards calls that allocate memory. We repeat 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 objects at runtime and generates a summary of memory leaks when the program terminates. Note that LSan does not modify 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 intercepted 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 geomean *improvement of 32%* compared to using ASan for target acquisition. Likewise for the integer overflow in `freetype2`, we see that using the correct sanitizer which actually catches the bug (i.e., UBSan) for target acquisition improves the performance significantly, finding the bug in 20 hours rather than 47 hours.

As shown in Table 5, there is a stark contrast between sanitizers used for target acquisition. We run a number of applications with known bugs of a certain type, while using three different sanitizers (ASan, UBSan, and TySan) to automatically find targets. Note that triggering the bugs requires sanitizers also (as the bugs usually do not crash the program). To eliminate the variance caused by overhead of each sanitizer, 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 allows ParmeSan to find bugs faster.

For example, CVE-2018-13785 is a hard-to-trigger inte-

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

Table 5: Bugs found by ParmeSan using different sanitizers in the analysis stage. ✓ in targets, bug found; ✗ not in targets, bug found; For the memory leak (ML) bugs we also show the performance of LeakSanitizer.

ger overflow in libpng. Here we see the most significant improvement as result of selecting the right sanitizer. Specifically, using UBsan, we trigger the bug in an average time of 32 minutes, but using the other sanitizers, ParmeSan does not consider the site triggering the bug as a target, and therefore takes a significantly longer time to find the bug, while using the right sanitizer for target acquisition improves the performance by an order of magnitude.

For the use-after-free bug in `pcre2`, both ASan and TySan 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 libxml), 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-

Prog	Version	Bugs	NEUZZ		QSYM		Angora		ParmeSan	
			1h	24h	1h	24h	1h	24h	1h	24h
OSS Fuzz [39]										
curl	54c622a	1	0	0	0	0	0	0	0	1
json-c	ddd0490	0	0	0	0	0	0	1	1	1
libtiff	804f40f3	1	0	0	0	0	0	1	1	1
libxml2	1fbcf40	2	0	0	0	0	0	1	1	2
libpcap	c0d27d0	1	0	0	0	0	0	1	1	1
OpenSSL	6ce4ff1	1	0	0	0	1	0	1	1	1
ffmpeg	9d92403	0	0	0	0	0	0	0	0	0
harfbuzz	b21c5ef	0	0	0	0	0	0	0	0	0
libpng	3301f7a1	0	0	0	0	0	0	0	0	0
Targets from prior work [3, 12, 32]										
jhead	3.03	2	0	2	0	2	2	2	2	2
pbc	0.5.14	37	9	9	2	12	10	29	23	37
protobuf-c	1.3.1	1	0	0	0	0	1	1	1	1

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.

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. Hawkeye [8] improves upon the ideas in AFLGo by supporting indirect calls using static alias analysis. While Hawkeye 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 compartments 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 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 reaching 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 also 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 (ONR) 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

[1] DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html. Online; accessed 30-March-2019.

[2] UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Online; accessed 30-March-2019.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Network and Distributed System Security Symposium (NDSS 2019)*, 2019.

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

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM.

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, volume 8, pages 209–224, 2008.

[7] Hongxu Chen, Yuekang Li, Bihuan Chen, Yinxing Xue, and Yang Liu. Fot: a versatile, configurable, extensible fuzzing framework. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 867–870. ACM, 2018.

[8] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108. ACM, 2018.

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

[10] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. In *ACM Conference on Computer and Communications Security (CCS)*, London, UK.

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

[12] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security*

Symposium (USENIX Security 19), pages 1967–1983, Santa Clara, CA, August 2019. USENIX Association.

[13] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 144–155. ACM, 2016.

[14] LLVM Developers. TySan: A type sanitizer. https://lists.llvm.org/pipermail/llvm-dev/2017-April/111766.html, 2017. Online; accessed 19-March-2019.

[15] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[16] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 60–71, Piscataway, NJ, USA, 2019. IEEE Press.

[17] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[18] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 474–484. IEEE Computer Society, 2009.

[19] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. DyTa: Dynamic Symbolic Execution Guided with Static Verification Results. pages 992–994, 05 2011.

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

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

[22] Inc. Google. fuzzer-test-suite. https://github.com/google/fuzzer-test-suite, 2018. Online; accessed 30-March-2019.

[23] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 49–64, Washington, D.C., 2013. USENIX.

[24] Mark Harman. Automated test data generation using search based software engineering. In *Proceedings of the Second International Workshop on Automation of Software Test*, page 2. IEEE Computer Society, 2007.

[25] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[26] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineer-ing*, 2018.

[27] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 1–13, Santa Clara, CA, July 2017. USENIX Association.

[28] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637. ACM, 2017.

[29] Paul Dan Marinescu and Cristian Cadar. KATCH: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245. ACM, 2013.

[30] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[31] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[32] Trail of Bits. ProtoFuzz: A Protobuf Fuzzer. https://blog.trailofbits.com/2016/05/18/protofuzz-a-protobuf-fuzzer/, 2016. Online; accessed 31-January-2019.

[33] Mathias Payer, Antonio Barresi, and Thomas R Gross. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164. Springer, 2015.

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

[35] V. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury. Hercules: Reproducing crashes in real-world application binaries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 891–901, May 2015.

[36] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium (NDSS)*, February 2017.

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. 2012.

[38] Kostya Serebryany. Sanitize, fuzz, and harden your C++ code. In *USENIX Enigma*, 2016.

[39] Kostya Serebryany. Oss-fuzz-google's continuous fuzzing service for open source software. 2017.

[40] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient fuzzing with neural program smoothing. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[41] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *ACM Sigplan Notices*, volume 50, pages 473–486. ACM, 2015.

[42] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: sanitizing for security. *arXiv preprint arXiv:1806.04355*, 2018.

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

[44] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *IEEE Symposium on Security and Privacy (SP)*, pages 866–879. IEEE, 2015.

[45] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy (SP)*, pages 497–512. IEEE, 2010.

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

[47] Michal Zalewski. American Fuzzy Lop: a security-oriented fuzzer. `http://lcamtuf.coredump.cx/afl/`, 2010. Online; accessed 31-January-2019.

[48] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. A platform for secure static binary instrumentation. In *ACM SIGPLAN Notices*, volume 49, pages 129–140. ACM, 2014.

A Additional results

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 fuzzer-test-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,

Prog	Type	Runs	ParmeSan		No lazysan		No pruning		No dyncfg	
boringssl	UAF	10	1850	25m	1850	37m	2503	47m	2520	51m
c-ares	BO	10	200	1s	200	1s	260	1s	200	1s
freetype2	IO	5	49320	43h	49320	46h	✗	✗	✗	✗
pcre2	UAF	10	8761	8m	8761	12m	29036	14m	10531	12m35s
lcms	BO	10	540	41s	540	1m10s	2990	2m10s	758	1m40s
libarchive	BO	10	4123	13m	4123	18m	6001	20m	5833	21m
libssh	ML	10	123	50s	123	31s	304	1m15s	285	55s
libxml2	BO	10	2701	11m	2701	17m	5066	20m	5123	23m
libxml2	ML	10	2554	17m	2554	15m	7580	22m	7966	25m
openssl-1.0.1f	BO	10	543	3m4s	543	4m30s	700	5m	610	4m52s
openssl-1.0.1f	ML	10	709	37s	709	40s	719	42s	713	42s
proj4	ML	10	80	1m26s	80	1m30s	83	1m40s	80	1m30s
re2	BO	10	3267	12m35s	3267	17m10s	3920	20m13s	3450	18m21s
woff2	BO	10	83	8m	83	13m	91	20m	83	13m
woff2	OOM	10	49	12s	49	19s	50	20s	49	19s
Geomean diff					**+0%**	**+25%**	**+19%**	**+28%**	**+17%**	**+34%**

Table 7: Impact of different components of ParmeSan on branch coverage and time-to-exposure of the bug.

Prog	Type	Run time		Compile time	Targets		
		DFA	+dyncfg	Target acquisition	ParmeSan	No c.b. pruning	No pruning
boringssl	UAF	2%	3%	200%	51	51	253
c-ares	BO	5%	5%	170%	21	21	36
freetype2	IO	5%	5%	170%	730	950	8538
pcre2	UAF	2%	2%	190%	1856	2051	21781
lcms	BO	0%	1%	140%	95	98	785
libarchive	BO	1%	1%	170%	273	340	1431
libssh	ML	3%	3%	180%	55	45	229
libxml2	BO	1%	1%	210%	670	751	5131
libxml2	ML	2%	2%	210%	670	751	5131
openssl-1.0.1f	BO	1%	1%	240%	43	39	304
openssl-1.0.1f	ML	1%	1%	240%	43	39	304
proj4	ML	3%	3%	140%	18	15	41
re2	BO	1%	1%	160%	295	370	2129
woff2	BO	1%	2%	180%	24	20	33
woff2	OOM	10%	10%	180%	24	20	22
Geomean		**2%**	**2%**	**183%**	**112** (+0%)	**108** (-3.5%)	**716** (+539%)

Table 8: Run-time and compile-time overhead introduced by the individual ParmeSan components.

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.

We fuzz 4 applications where we artifically demote (a random selection of) of the direct calls to indirect calls (with 2 dummy call targets added) and obtain the targets using the ParmeSan pipeline (with ASan), 3 applications where we demote direct calls and manually target the bug, and finally run the whole ParmeSan pipeline (with ASan) opn 3 real-world applications with a large number of indirect calls. The results for these three experiments can be found in Table 9. Overall, we see that the dynamic CFG component has a higher impact if there are indirect calls on the path to the bug to be found (e.g., in `libjpeg-turbo`). We also kept track of how much time is spent on the dynamic CFG component. Overall the overhead is negligible in most cases, accounting for less than 3% of the total execution time (as shown in Table 8).

A.1.2 Comparison against SAVIOR

For the sake of completeness, we include Table 10, which shows how ParmeSan compares against Angora and SAV-

Prog	Calls demoted	Mean. TTE		p-val
		no dyncfg	dyncfg	
base64	5	55s	54s	0.19
who	10	2m32s	2m21s	**0.03**
uniq	8	48s	22s	**0.005**
md5sum	15	8m34s	6m32s	**0.007**
Manual targeting				
libjpeg-turbo	30	43m	11m	**0.004**
libpng	20	1m29s	21s	**0.006**
libpng	20	10s	10s	0.06
freetype2	5	1s	1s	0.09
Real-world programs				
httpd	0	10s	1s	**0.003**
cxxfilt	0	1m45s	1m5s	**0.02**
boringssl	0	51m	37m	**0.005**

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.

	Angora	SAVIOR	**ParmeSan**	
			ASan	`lava_get()`
base64	48	48	48	48
md5sum	59	59	60	60
uniq	29	29	29	29
who	2295	2357	2320	2353

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 targetting `lava_get()` (replicating the setup described in [11]).

Prog	Targets (pre-prune)	Targets (post-prune)	Bugs	Bugs Found		
				1m	1h	24h
base64	1950	212	44	48	48	48
md5sum	1639	101	57	31	59	60
uniq	1832	193	28	29	29	29
who	2120	385	2136	1544	1957	2353

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.

IOR on the well-known (but what might be considered outdated) LAVA-M dataset. We include this table to be able to show a head-to-head comparison against SAVIOR. We replicate the setup used by SAVIOR in [11], where the targets are acquired in a manual way (i.e., explicitly targeting the inlined calls to `lava_get()`), rather than using sanitizers for target acquisition, and use 3 fuzzing instances in parallel. Overall, the results for ParmeSan and SAVIOR are comparable, with the exception of `md5sum`, where ParmeSan finds one more hard-to-trigger bug (unlisted bug #499) and `who`, where SAVIOR is able to trigger two more bugs. We hypothesize that ParmeSan is able to trigger the `md5sum` bug due to its ability to execute more test cases per second, while SAVIOR is better at finding the remaining two bugs in `who` due to its symbolic execution-based constraint solving strategy. Moreover, with ParmeSan, we were able to reproduce the very same results on LAVA when using a single fuzzing instance (and CPU core), suggesting ParmeSan's fuzzing-only strategy can provide results comparable to SAVIOR's constraint solving-assisted strategy but with less resources. We also include the results for using ASan for targeting.