SoK: Where to Fuzz?
Assessing Target Selection Methods in Directed Fuzzing

Felix Weissberg[1], Jonas Möller[1], Tom Ganz[2], Erik Imgrund[2], Lukas Pirch[1], Lukas Seidel[3], Moritz Schloegel[4], Thorsten Eisenhofer[1], Konrad Rieck[1,5]

[1]Technische Universität Berlin & BIFOLD
[2]SAP Security Research [3]Binarly
[4]CISPA Helmholtz Center for Information Security
[5]Technische Universität Wien

ABSTRACT

A common paradigm for improving fuzzing performance is to focus on selected regions of a program rather than its entirety. While previous work has largely explored *how* these locations can be reached, their selection, that is, the *where*, has received little attention so far. In this paper, we fill this gap and present the first comprehensive analysis of target selection methods for fuzzing. To this end, we examine papers from leading security and software engineering conferences, identifying prevalent methods for choosing targets. By modeling these methods as general scoring functions, we are able to compare and measure their efficacy on a corpus of more than 1,600 crashes from the OSS-Fuzz project. Our analysis provides new insights for target selection in practice: First, we find that simple software metrics significantly outperform other methods, including common heuristics used in directed fuzzing, such as recently modified code or locations with sanitizer instrumentation. Next to this, we identify language models as a promising choice for target selection. In summary, our work offers a new perspective on directed fuzzing, emphasizing the role of target selection as an orthogonal dimension to improve performance.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; **Systems security**; • **General and reference** → Surveys and overviews.

KEYWORDS

Directed Fuzzing, Software Security, Target Selection, Software Metrics

ACM Reference Format:
Felix Weissberg, Jonas Möller, Tom Ganz, Erik Imgrund, Lukas Pirch, Lukas Seidel, Moritz Schloegel, Thorsten Eisenhofer, and Konrad Rieck. 2024. SoK: Where to Fuzz? Assessing Target Selection Methods in Directed Fuzzing. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3634737.3661141

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASIA CCS '24, July 1–5, 2024, Singapore, Singapore
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0482-6/24/07.
https://doi.org/10.1145/3634737.3661141

1 INTRODUCTION

Fuzzing has been a thriving area of security research in recent years. Numerous techniques have been proposed to improve the efficacy of fuzzers, ranging from concepts to increase efficiency [3, 17, 61, 64] and mitigate roadblocks [4, 5] to application-specific testing strategies [16, 56, 59]. This research encompasses a large variety of technical contributions inspired from software engineering, compiler design, and software security, yet one general paradigm stands out as a key concept in many of the approaches to improving fuzzing performance: *target selection.*

Rather than treating all code locations equally, many approaches guide the fuzzer towards areas more likely to contain defects, such as those involving memory accesses, insecure API calls, or recent patches [32, 49, 83]. Similarly, several fuzzers direct the testing explicitly to a chosen set of locations to improve efficiency in different stages of software development [8, 15, 34]. Although these approaches fundamentally differ in *how* they reach these regions and guide the fuzzer, they all share the concept of focusing on selected target locations rather than uniformly covering the program as traditional fuzzers [66].

Surprisingly, the analysis of suitable targets for a fuzzer, that is, the *where* of the paradigm, has received little attention so far. While some fuzzers incorporate a fixed strategy for locating targets, such as focusing on sanitizers [83], code changes [49], or buffer operations [32], several approaches for directed fuzzing leave the target selection to the practitioner. As a result, it is currently unclear which selection method performs best in practice, and the simple question—*where to fuzz?*—is still unexplored.

In this paper, we aim to bridge this gap and present the first comprehensive analysis of *target selection methods* for directed fuzzing. To this end, we conduct a literature review of 25 papers concerned with directed fuzzing published at top-tier security and software engineering conferences over the past five years. We identify two dimensions that characterize the selection of targets: (1) the information source and (2) the selection mechanism. For example, some selection methods are based on analyzing binary code, while others require the source code or further external information. Similarly, we can categorize the selection mechanisms into metric-based and pattern-based heuristics, identifying interesting code locations for the fuzzer to inspect.

Based on this systematization, we proceed to compare different methods for target selection. To avoid evaluating the efficacy of fuzzers rather than their targets, we conduct this comparison in isolation. That is, we model target selection methods as abstract

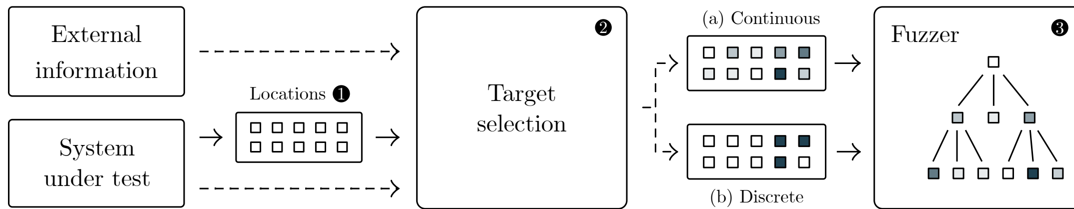


Figure 1: Target selection. The initial step in target selection is the extraction of code locations from the SUT (Step ❶). The granularity of this extraction depends on the specifics of the selection method and could, for example, be on a function-level or basic-block level. After extraction, the locations and (optionally) the SUT or external information (e.g., code change timestamps) are forwarded to the target selection method. This method then assigns a score to each location (Step ❷), which is either (a) continuous or (b) discrete. Finally, the fuzzer utilizes the annotated code locations for guidance (Step ❸).

scoring functions that retrieve a set of code units and assign scores to them, indicating their "interestingness" for the particular fuzzing setup. This approach is applicable to any selection method and differs only in (1) which code units are provided and (2) how they are ranked according to the underlying scoring functions. As a result, we are able to formulate our evaluation as an information retrieval task and assess the methods "in vitro" using corresponding performance measures.

For our evaluation, we assemble a dataset of more than 1,600 crashes from 97 software projects of OSS-Fuzz [62], which to the best of our knowledge represents the largest corpus of reproducible crashes to date. These crashes serve as ground truth and allow for quantitatively measuring how well a target selection method and the underlying scoring function can retrieve code that actually triggers crashes during a fuzzing campaign. By measuring this retrieval performance in different settings, we can draw general conclusions about target locations and derive recommendations for improving directed fuzzing, independent of specific testing strategies.

Our evaluation uncovers different insights for target selection: First, we observe that simple software metrics, such as vulnerability scores of the Leopard framework [24], significantly outperform all other approaches. This holds true across all types of crashes and indicates that advanced methods for target selection are not yet able to surpass simple numerical metrics. Second, we identify large language models for code, such as CodeT5+, as promising alternatives that almost reach the efficacy of software metrics.

In summary, our work opens a new perspective on directed fuzzing by decoupling the fuzzing mechanism (how) from the selection of target locations (where). We are the first to observe that selection methods differ significantly when studied in isolation, and that some strategies, such as software metrics, consistently outperform other methods. Given the simple nature of these metrics, we argue that developing better selection methods is a promising path for future research and that machine learning models can potentially serve as a means to success in this area.

Contributions. In summary, we make the following major contributions in this work:

(1) *Systematization of target selection.* We conduct the first systematic analysis of target selection methods for directed fuzzing. For this, we analyze a total of 25 papers related to target selection published between 2018–2023 at top security and software engineering conferences.
(2) *Large-scale evaluation.* We model the task of target selection as an information retrieval problem and assemble a dataset of more than 1,600 reproducible crashes as ground truth. This is the largest available dataset of this kind, and we provide it publicly as part of this paper's artifacts.
(3) *Insights for target selection in fuzzing.* Based on the results of our analysis, we provide insights and recommendations for target selection in different fuzzing contexts.

2 FUZZING SPECIFIC CODE LOCATIONS

The ultimate goal of fuzzing is to uncover bugs. When testing unknown software, the underlying defect distribution, i.e., the number and location of defects, is naturally unknown. In this case, code coverage has proven an excellent proxy metric: after all, we cannot find flaws in code that has not been executed by the fuzzer. In fact, recent research has found a strong correlation between code coverage and software defects [10]. However, the fuzzer performing best in terms of coverage may not necessarily be the one finding the most bugs. Still, using code coverage as *feedback* for guidance has become the de-facto standard of modern fuzzing, offering excellent exploration capabilities in absence of knowledge of the underlying defect distribution.

In certain scenarios, however, we may know, or at least can assume, the (potential) location of defects. For example, new and untested code is more likely to contain bugs than well-tested software [81]. Similarly, a static analysis tool can pinpoint specific code locations as potentially buggy. In such cases, *guiding* the fuzzer towards these locations may help to uncover defects faster than when indifferently exploring all parts of the program. This insight lead to the introduction of *directed fuzzers*: Traditionally, these fuzzers accept one or more code locations, for example, addresses in the binary or source code lines, that a fuzzer should target during execution. While some fuzzers directly turn their focus towards these targets, others initially explore the program similarly to regular fuzzers before switching to an exploitation phase where they focus on the desired target locations [8].

In this work, we consider a wider definition of directed fuzzing: We not only include traditional approaches that accept predefined

target locations but also consider any approach that uses some metric (beyond coverage feedback) to guide the fuzzer to specific locations. A prime example of such an approach is *regression greybox fuzzing* [81], where testing is directed towards recently changed code rather than user-specified targets. While different to traditional directed fuzzing that uses a fixed set of targets, this technique still *directs* the fuzzer to specific locations. The difference merely lies in the *target selection* and the type of *scoring* used to measure the "relevance" of a given code location. In other words, the score of code locations tells the fuzzer whether to prioritize them during fuzzing. Figure 1 outlines the overall process enabling the guidance of the fuzzer. Before discussing the individual steps and focusing on the *target selection*, we first motivate its relevance towards improving the probability of finding a crash. Following this, we examine the two different types of scoring used in directed fuzzing in more detail.

Relevance of target selection. When examining prior works that present directed fuzzers, we can make two observations. First, these studies typically demonstrate—as part of their experiments—that their directed fuzzing approach outperforms an undirected coverage-guided fuzzer that serves as a baseline [2, 7, 15, 53, 74, 78]. Second, they show their fuzzers are in fact directed; that is, they gradually steer the fuzzing process towards the provided targets [7, 15, 74]. If we assume all targets are unrelated to crash sites, a directed fuzzer will spend resources to focus on code that contains no bugs, meaning the directed fuzzer is unlikely to outperform a coverage-guided one in this case. Thus, we conclude that selecting relevant targets is indeed a crucial step in directed fuzzing to increase the probability of finding crashes. In other words, not only implementation details such as fuzzing throughput determine the fuzzer's success but also the target selection. Despite its relevance to the success of directed fuzzing, we find this aspect is often overlooked and not considered individually. With this in mind, we now take a closer look at the two types of scoring mechanisms that enable a selection of targets.

Discrete scoring. The first type of scoring assigns a *discrete* value, usually either 0 or 1, to each location. Hence, it differentiates between relevant locations and irrelevant ones. This scoring type closely resembles most tools traditionally referred to as directed fuzzers: They accept a set of target locations, marking them as relevant, whereas every other code location is considered irrelevant. The target selection decides upon the metric according to which target locations are chosen. During execution, the *distance* to relevant code locations is then used as a proxy score that allows to indirectly rank and compare locations.

Continuous scoring. The second type of target selection assigns a *continuous* value to all code locations: This score can, for example, be based on the last time this location has been modified [81], the number of sanitizer primitives it contains [83], or a software metric describing its code complexity [24]. As each code location is assigned its own score, these approaches usually do not need a proxy metric, such as the distance to a target location, and the fuzzer can iteratively optimize over the scores of the visited code regions during its operation.

Comparison of scoring functions. The two types of scoring functions lead to a different effectiveness of target selection. To examine this difference, let us consider a program that contains a single defect. With continuous scoring, the probability that this bug is triggered increases steadily with the quality of the scores, as a high quality scoring means that the defective locations are assigned significantly higher scores than non-defective ones.

In contrast, when we consider a discrete scoring, we usually focus on a small number of targets. In this case, a low-quality scoring may not flag the error location as relevant, and hence it may become unlikely that the fuzzer reaches the bug at all, as it is effectively steered towards non-defective locations. However, once the location is marked, our chances increase noticeably. When evaluating target selection methods in Section 5, we take this difference into account.

In comparison, discrete scoring provides more versatility: Regardless of how the target set was derived, guiding a fuzzer using the distance to its locations will work, providing greater versatility compared to scoring each code locating using a specific metric. Changing the metric according to which targets are selected is supported by design: Often, these tools allow the user to specify arbitrary code locations. On the other side, all selected locations are treated equally by discrete scoring functions. If two of these targets are not equally relevant, we lose information, as individual locations cannot be ranked differently. In contrast, continuous scores do not suffer from this problem and can differ between targets on a more fine-granular basis.

In short, we can distinguish target selection approaches based on the score they assign to their targets, with each type having advantages and disadvantages.

Overview. With the two types of scoring in mind, we can focus on their place in the overall fuzzing process. Figure 1 provides an overview of the general flow: Based on external information known a priori or information extracted from the System Under Test (SUT), we know *what* code to target. Our target selection receives this metric and the SUT (Step ❶). Based on the available information, the *target selection* (Step ❷) can be performed by assigning a *discrete* or *continuous* score to each code location. Finally, the fuzzer tests the SUT (Step ❸) while focusing on the targets.

3 SYSTEMATIZATION OF TARGET SELECTION

Selecting good targets is crucial to the success of directed fuzzing. Yet, we find that this aspect has surprisingly received little attention in fuzzing research so far. Consequently, before turning towards a comparative analysis of different strategies for locating interesting code, we first conduct a comprehensive literature review on target selection methods used currently for directed fuzzing.

Literature. For our review, we investigate papers published at top security and software engineering venues between 2018–2023. In particular, we collect all papers from the following A* conferences [54]: ASE, FSE, ICSE, CCS, NDSS, USENIX Security, and S&P. After filtering out papers not related to fuzzing, this leaves us with 289 fuzzing papers. We then automatically identify papers focusing on *directed fuzzing* by filtering out any paper that does not contain the word "directed" at least three times. This heuristic is grounded in the assumption that publications in a field are likely to either name the

field multiple times or at least mention it when comparing against prior works. Using this process, we end up with 31 publications. Manually analyzing all of them yields 25 papers that deal with directed fuzzing under our definition.

Review method. To distinguish between different target selection techniques, we review all papers in-depth and distill the working principle for locating targets. As result of this process, we arrive at four characteristics to categorize selection methods. First, we consider the source from which the information originates that is used to direct the fuzzer. We refer to this as the *information source*. Second, we distinguish whether the underlying scoring function is *discrete* or *continuous* as discussed previously. We label this as the *scoring type* of the selection method. Third, we distinguish between different levels of granularity of the target selection method. We denote this as the *granularity*. Lastly, we differentiate target selection techniques with respect to what their scoring mechanism is based on. We refer to this property as the target selection's *scoring mechanism*.

Analysis results. The results of our literature review are shown in Table 1. For each publication, we indicate whether a paper put a focus on the target selection (denoted as ✓) or if it exclusively used targets obtained from third parties, such as previous literature or stack traces of crashes from a bug tracker (denoted as ✗). We further record whether the employed target selection was continuous (C) or discrete (D). In the following, we shift our focus to the information sources, scoring mechanism and granularity used within the examined publications.

Table 1: Overview of target selection methods. The *Where?* column denotes if a publication puts a focus on the target selection method (✓) or if it exclusively relied on a target selection from previous work or reproduction tasks (✗). Column *Scoring type* records if a discrete target selection methods (D) or a continuous one (C) is used. The granularity of the target selection method is categorized in *Granularity*: instructions (I), basic blocks (B), statements (S), lines (L), and functions (F). The *Source* and the *Scoring* column provide further details about the origin of the additional information and utilized core method of the target selection, respectively.

					Source			Scoring	
Paper	Venue	Where?	Scoring type	Granularity	Source code	Binary code	External info	Metrics-based	Pattern-based
SelectFuzz [46]	SP'23	✗	D	L	○	○	●	○	●
ODDFuzz [13]	SP'23	✓	D	F	●	○	○	○	●
StrawFuzzer [77]	SP'22	✓	D	F	●	○	○	○	●
GREBE [45]	SP'22	✗	D	S	○	○	●	○	●
BEACON [34]	SP'22	✗	D	L	○	○	●	○	●
She et al. [65]	SP'22	✓	C	S	●	○	○	●	○
SAVIOR [19]	SP'20	✓	D	B	○	◐	○	○	●
CollAFL [31]	SP'18	✓	C	B	○	●	○	●	○
DAFL [39]	SEC'23	✗	D	L	○	○	●	○	●
DDRace [76]	SEC'23	(✓)	D	I	○	◐	○	○	●
FishFuzz [79]	SEC'23	✓	D	F	○	●	○	○	●
AmpFuzz [42]	SEC'22	✓	D	F	●	○	○	○	●
Lee et al. [43]	SEC'21	✗	D	L	○	○	●	○	●
FuzzGuard [82]	SEC'20	✗	D	L	○	○	●	○	●
ParmeSan [83]	SEC'20	✓	D	B	○	●	○	○	●
Jiang et al. [37]	NDSS'22	✓	D	F	○	○	●	○	●
TortoiseFuzz [70]	NDSS'20	✓	C	B	○	●	●	●	○
MC^2 [63]	CCS'22	✗	D	L	○	●	●	○	●
AFLChurn [81]	CCS'21	✓	C	L	○	○	●	○	●
VulScope [23]	CCS'21	✗	D	F	○	○	●	○	●
Hawkeye [15]	CCS'18	✗	D	L	○	○	●	○	●
WindRanger [25]	ICSE'22	✗	D	L	○	○	●	○	●
AFL_{LTL} [51]	ICSE'22	✓	D	S	○	○	●	○	●
Wüstholz et al. [74]	ICSE'20	✗	D	L	○	○	○	○	○
Leopard [24]	ICSE'19	✓	C	F	●	○	○	●	○

The scoring mechanism landscape. From our literature review, we observe two broad categories of scoring mechanisms: one based on metrics and one on patterns. The metrics-based category includes methods that utilize code metrics to score individual code locations. In contrast, the pattern-based target selection category utilizes heuristics for this purpose.

Out of the 25 directed fuzzing approaches, 14 put a focus on the target selection method (✓in the *Where?* column of Table 1). From these 14, ten are pattern-based and use heuristics to select interesting target locations. For example, ODDFuzz [13] selects deserialization methods in Java as targets for a directed fuzzing approach, StrawFuzzer [77] uses data storing instructions as targets to increase the memory footprint of Android services and cause the system to crash, and AmpFuzz [42] targets network-related functions as part of their approach to find amplification vectors for DDoS attacks. All of these heuristics have in common that they focus on finding a specific type of erroneous behavior. Another heuristic employed by three of the approaches, ParmeSan [83], SAVIOR [19], and FishFuzz [79], covers a broader range of potential bugs: Their approach to the target selection is based on the idea that sanitizer instrumentation can function as an indicator for the relevance of a code location to a directed fuzzer. After all, sanitizer instrumentation is added at locations where a bug might occur.

Instead of heuristics, five approaches use a metrics-based selection method. In particular, TortoiseFuzz [70] and CollAFL [31] focus on the number of memory accesses in their target selection method. In addition, TortoiseFuzz augments this information with the number of security related functions, which they identified by crawling the pages referenced from CVE descriptions. Both the target selection used by She et al. [65] and the one presented by Leopard [74] are based on code metrics. The former uses a graph-based metric that assigns scores to code locations based on how many other uncovered code regions could potentially be reached from it. The latter uses two code metrics, a structural complexity metric and what the authors refer to as "vulnerability metrics", which revolve around properties of a code region, such as the number of pointer arithmetic operations or the number of nested control structures.

The information source landscape. The sources providing input to the target selection methods can be broken down into three origins: source code, binary code, and external information.

The results of our literature survey show that each information source is used by metrics-based as well as pattern-based methods. However, the specific information source is, unsurprisingly, determined by the individual method. Certain information is only available at the source code level, while others are accessible only on the binary code level. For instance, *sanitizer instrumentation* is not present at the source code level, such that the approaches we examined either resorted to the binary code level or LLVM IR for this purpose. We denote the latter with half-filled circles (◐) in Table 1. The same holds true for *memory access instructions* [31, 76]. On the other hand, the code metrics used by Leopard [24] require source code as input. In contrast, AFLChurn [81] makes use of the information when a code location was last changed, and TortoiseFuzz [70] uses information crawled from pages referenced by the CVE database. In both cases, the information is neither present in the source code nor in the compiled binary but acquired from external sources.

The granularity. Unlike other systematic literature reviews which focus on the fuzzing approaches themselves [44, 68], we characterize the methods used for selecting targets. Consequently, in our setting, the granularity refers to the target selection method rather than the characteristics of the proposed directed fuzzer. In other words, while certain fuzzers, such as StrawFuzzer [77] or Hawkeye [15] operate on a basic block granularity, the employed target selection methods provide targets with a function respectively source code line granularity.

In our analysis, we categorize all target selection methods into five distinct levels of granularity: instructions (I), basic blocks (B), statements (S), lines (L), and functions (F). While instruction and basic block granularity only concern methods using the binary code as information source, function granularity can apply to both binary and source code. Line and statement granularity refer to source code lines and individual statements within a line, respectively. We find that most papers that introduce a new target selection method use a function level granularity. On the other hand, papers that do not introduce a new target selection method are dominated by line level selection methods. This mostly stems from the fact that these papers resort to the source file and line number obtained from tracebacks of crashes as the targets to evaluate their fuzzer.

4 ANALYSIS FRAMEWORK

Although numerous target selection mechanisms have been proposed, they rarely receive special attention during the evaluation of the directed fuzzers for which they were designed. Instead, fuzzer evaluations commonly compare different tools against each other so that observed differences cannot confidently be attributed to individual components such as the target selection. Furthermore, many evaluations (re-)use the same set of targets (e.g., the dataset AFLGo [8] presented), thereby measuring performance differences between the newly proposed technique and existing tools, without scrutinizing the strategy for selecting these targets in the first place nor reflecting the various bug classes or target SUTs a selection should perform well on.

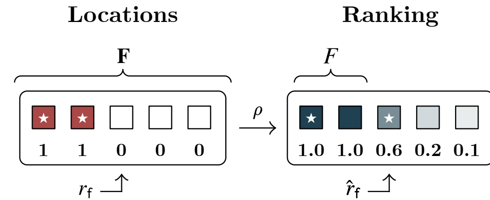


Figure 2: Overview of a retrieval. We compute a ranking using target selection method ρ which assigns each function $\mathrm{f} \in \mathbf{F}$ a relevance score $\hat{r}_{\mathrm{f}}$. To measure the quality of target selection method, we compute the $NDCG_k$ for a retrieval F with cardinality k. As ground truth, we use the oracle O to assign relevance scores r_{f} to each location $\mathrm{f} \in \mathbf{F}$.

Requirements. Therefore, we identify two requirements for conducting a thorough and systematic comparison of target selection mechanisms.

R1. We need a suitable method by which we can compare different target selection techniques in isolation, independent of a specific fuzzer (→ Section 4.1).

R2. For comparative purposes, a comprehensive ground truth is necessary, ideally representing many different bug classes across a large variety of real-world code (→ Section 4.2).

4.1 Target Selection as Information Retrieval

As a first step for addressing these requirements, we present our method for assessing the effectiveness of target selection methods. Our approach is based on the observation that target selection fundamentally resembles an *Information Retrieval* (IR) problem: that is, target selection can be conceptualized as the *retrieval* of relevant target locations in a software project using a scoring mechanism. In this formulation, targets are *relevant* with regard to the directed fuzzing process, if they benefit the effectiveness of a fuzzer in terms of uncovering new bugs. A fuzzer can use different detection mechanisms to identify whether it triggered a bug; the most common one used by virtually all fuzzers are program crashes, which we focus on subsequently. In other words, targets are relevant if they can point to a code location at which the fuzzer finds a program crash. Similarly, a target can be considered *irrelevant* if it does not contribute to the discovery of a defect and the fuzzer essentially wastes time exploring it. Consequently, a selection methods performs better if it assigns higher scores to relevant code during fuzzing.

Considering target selection as an information retrieval problem yields three key advantages: First, it allows the evaluation of selection methods agnostic to a particular implementation of a directed fuzzer. Second, it enables to capture different requirements for discrete and continuous target selection methods in form of discrete and continuous retrieval. Third, we can employ standard evaluation measures from the information retrieval domain to compare and assess the selection on a well-established ground.

In this work, we focus on a function-level granularity as the input for the target selection method. Intuitively speaking, the optimal target selection method would, thus, precisely *retrieve* the subset F of all functions $\mathbf{F}$ within a given software project that lead to a crash. However, given that the exact size of subset F is typically unknown,

we instead consider the k highest-rated functions as determined by the target selection's scoring method. This scoring can be described as a mapping ρ from functions f to relevance scores r:

$$\rho\colon \mathbf{F} \longrightarrow \mathbb{R}^{+}, \quad \mathrm{f} \mapsto r\,.$$

For simplicity, we assume that any external information required for the scoring is embedded into the input to ρ. Based on these scores, we then compute a ranking over $\mathbf{F}$ and retrieve the subset F as the first k entries in this ranking. Subsequently, subset F is returned as the output of the target selection.

Ranked retrieval measure. As common in information retrieval, we distinguish between relevant and irrelevant objects, i.e., whether a bug is present or not. Thus, to assess the effectiveness of a particular retrieval F, two requirements need to be taken into account: (1) the number of relevant functions in the retrieval, and (2), their respective positions in the ranking. Metrics like *precision@k* or *recall@k* are often used for the former, but fall short in capturing the quality of the ranking. While this would not be a problem for discrete scoring functions, we would lose information for continuous ones. Therefore, we consider the *normalized discounted cumulative gain* (NDCG), a standard performance measure from the information retrieval literature, which accounts for both the relevance of a retrieved function and its rank.

To assign a relevance $\hat{r}_{\mathrm{f}}$ to each function $\mathrm{f} \in \mathbf{F}$, we assume there exists an oracle $\mathrm{O} : \mathbf{F} \to \{0, 1\}$, that assigns 1 to functions which appeared in the stack trace of a crash, and 0 otherwise. We can then use this oracle to assign each function a ground truth relevance score $\hat{r}_{\mathrm{f}} = \mathrm{O}(\mathrm{f})$. Functions which are likely to appear in the stack trace of every crash, such as the main function or functions introduced by sanitizers, are assigned a relevance 0, regardless.

Based on these scores and retrieval F with length k, we compute the NDCG_k in three steps: First, we compute the cumulative gain as the sum of the ground truth relevance scores $\hat{r}_{\mathrm{f}}$ of all retrieved functions $\mathrm{f} \in F$. This ensures the first requirement. To account for the second requirement, we reduce each relevance score proportional to the rank i of function f using a discount function (e.g., the binary logarithm [72]). Finally, we normalize the resulting gain to account for stack traces with different lengths. Therefore, we calculate the ideal discounted cumulative gain IDCG_k as in the previous two steps but assume a perfect ranking (i.e., all relevant functions are positioned at the top ranks). Formally, the performance measure is thus defined as

$$\mathrm{NDCG}_k \coloneqq \frac{1}{\mathrm{IDCG}_k} \sum_{i=1}^{k} \frac{\hat{r}_i}{\log_2(i+1)}\,.$$

A perfect $NDCG_k$ score of 1 indicates that the relevant functions were returned at the top ranks and a score of 0 implies that no relevant function was among the k retrieved ones. Using such a ranking-based measure rather than the scores directly enables a more robust comparison between different scoring methods. In particular, in this case, it does not matter whether one method generally assigns higher scores than another.

Matching policies. So far, we were operating under the implicit assumption that all functions in a given stack trace are relevant for a particular crash. However, this assumption is often inaccurate. In reality, it is generally challenging to pinpoint the exact *root cause* of a crash [6]. Consequently, we often lack precise knowledge about the specific function responsible for introducing a bug. Additionally, crashes might manifest only through particular sequences of function calls, and may not result from a single flawed function [81].

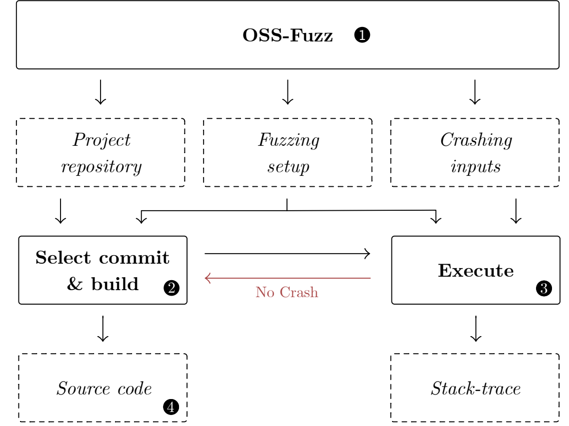


Figure 3: Dataset generation process. As basis for our analysis, we collect 1,621 reproducible crashes. We crawl OSS-Fuzz, which yields a crashing input and a fuzzing configuration for a project (❶). To reproduce the crash, we search for a commit of the project which crashes (❷) when executed under the input from OSS-Fuzz (❸). Once we reproduce a crash, we extract functions from the project's code and label them according to the stack trace (❹).

To account for this uncertainty, we opt for a middle ground by over-/ and underestimating our gain. Therefore, we introduce two matching policies:

Optimistic matching. First, we consider an optimistic matching policy that assigns a relevance score of 1 only to the *first* retrieved function from the stack trace. We denote this with NDCG^{+}. As a consequence, this *overestimates* the performance of a target selection method and serves as a upper bound in our analysis.

Pessimistic matching. Second, we consider a pessimistic policy that assigns a relevance score of 1 to *all* retrieved functions from the stack trace. We denote this with NDCG^{-}. This underestimates a target selection method and serves as a lower bound.

4.2 Crash Dataset

So far, we discussed how we can conceptualize and evaluate target selection methods from an information retrieval perspective. The only missing piece for our evaluation is a ground truth dataset composed of code locations annotated based on whether bugs are present or absent there.

A possible approach to obtain such data points is to fuzz various open-source projects, collect information about identified bugs, and trace back their location within the software project. Unfortunately, this process is very time-consuming and there is no guarantee that we will find all bugs in a given project, which would introduce false negatives to our analysis. As an alternative, we opt to establish our ground truth data by leveraging historical fuzzing campaigns, and, more precisely, crashes detected by the OSS-Fuzz [62] project. OSS-Fuzz has consistently applied fuzz testing to an extensive array of

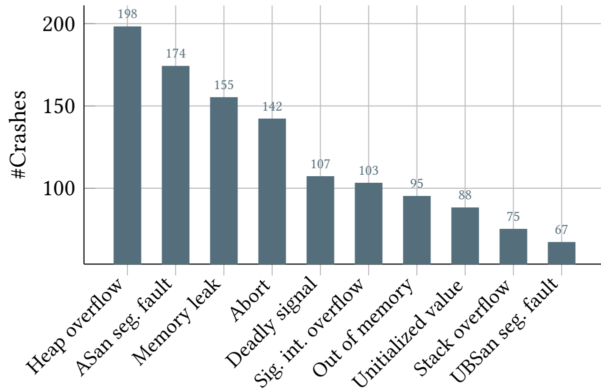


Figure 4: Crash types. We show the top ten crash types in our ground truth dataset.

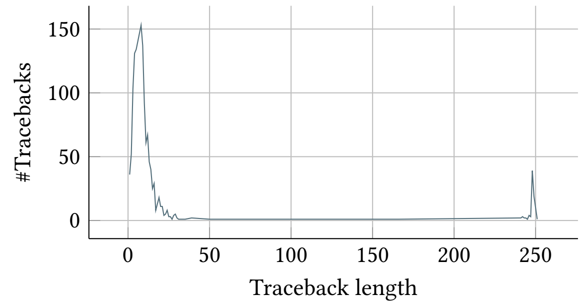


Figure 5: Stack trace lengths. We show the frequency of stack trace lengths from reproduced crashes. We observe the dominant peak at 7 functions per stack trace.

open-source projects since 2016 and publicly discloses their findings. While there is still no guarantee that OSS-Fuzz identified every bug, the massive amount of time spent on fuzzing these targets is likely to find a large majority of crashes reachable by a fuzzer.

For using crashing inputs identified from fuzzing for our ground truth, we require two things: (1) Crashes need to be reproducible to verify that they are correct, and (2) we need information about which specific functions are involved in the crash. A good approximation for this is provided by the stack trace of the crash. Thus, to utilize the data provided by OSS-Fuzz, we must address two primary challenges: First, OSS-Fuzz only releases the time at which a bug is reported and sometimes provides a regression range for when it was resolved but the exact commit of the software project is not disclosed. Second, and more importantly, OSS-Fuzz only shares detailed findings with authorized individuals (i.e., the project maintainers). In particular, the stack trace of a crash is not publicly accessible.

To remediate both issues, we pinpoint a commit at which the crash occurs, which in turn enables us to gather the relevant information for our ground truth. The details of this process are visually depicted in Figure 3 and further described in the following.

Data collection. We start our collection process by scraping the OSS-Fuzz issue tracker, which holds information about each crash, such as the project, reporting time, used fuzzer, sanitizer, and the crashing input (❶). We then select a commit close in time to the initial crash detection (❷). We rollback the project to this commit and build it with the configuration used by OSS-Fuzz to detect the crash. In case the project build fails, we retry the process by selecting an older commit. In some cases, there are broken references to external dependencies that we fix manually. Once a project is successfully built, we execute the crashing input (❸). If the crash can be observed, we collect a stack trace. If the program does not crash, we go back to ❷ and retry with an older commit.

After successfully recreating the crash and extracting stack trace information, we proceed to extract every function from the project's source code (❹). Many projects use pre-compiler directives to enable or disable certain features at compile time, which might include or exclude various code sections. Directly using the source code would thus pose a disadvantage to target selection methods working on the source code level: The source code may actually exhibit defective behavior that, however, will never be labeled as such by OSS-Fuzz, as the defective feature is always disabled via pre-compiler directives. To counteract this, we resort to post-pre-compiler code as the function representation for our dataset. For simplicity, we continue to refer to this code as source code.

Regarding the method's relevance oracle. The previous section's oracle O can be approximated by utilizing the information gathered from OSS-Fuzz. Specifically, we can use historic stack trace data and the mapping to the extracted source code to assign functions a score based on whether they were part of a stack trace of a crash found by OSS-Fuzz. This enables us to assemble a labelled corpus that we can further use to compare the retrieval performance of target selection methods to each another.

Dataset statistics. We have successfully reproduced 1,621 crashes across 97 C/C++ projects discovered by OSS-Fuzz between 2016 and 2023. We categorize crashes into 48 distinct crash types based on the report by the sanitizer used for identification. Figure 4 provides an overview of of the top ten crash types that contribute for approximately 75% of the crashes in the corpus, with heap overflows being the most prominent class. The distribution of crashes in the corpus, broken down by sanitizers, is further illustrated in Figure 6 showing that most crashes were identified by address sanitizer, which in total accounted for 62% of the crashes. The remainder of the crashes are identified by memory sanitizer and undefined behavior sanitizer, which make up 31% and 7% of the crashes respectively. Finally, we present the frequency distribution of stack trace lengths in Figure 5. We find the traceback length frequency shows a main peak at seven and a second, less pronounced, peak at 248. While a diverse set of crashes contributes to the former, the latter mostly consists of stack overflow crashes, that is, errors caused by reaching the stack limit.

The dominance of specific crash types and sanitizers used to identify the crashes should be taken into account for the comparison of the target selection methods.

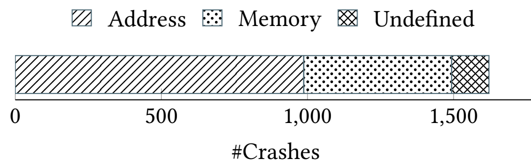


Figure 6: Number of crashes per sanitizer. We show the number of crashes in the corpus broken down by sanitizers.

5 COMPARISON OF TARGET SELECTIONS

With our analysis framework at hand, we can now dive into the comparison of target selection methods. Before we get started, however, we have to make a choice of the methods which we take into account.

5.1 Choice of Target Selection Methods

Before a fuzzer can be directed towards a specific location, we must make a suitable choice in terms of target selection. With Table 1 outlining the common techniques used for fuzzing, we have ample choices to select from. In the following, we desire to evaluate the quality of target selection methods. For an informed evaluation, we select three target selection approaches covering both core methods, pattern-based and metrics-based methods: For the former, we pick up the idea of using sanitizer instrumentation [19, 79, 83] and the idea of using recently modified code changes [81] for target selection. For the latter, we resort to the code metrics presented by Leopard [24].

To better put these methods used by fuzzing into context, we also include approaches no fuzzing paper has used yet. First, we turn to *static analysis*, which has found widespread adoption in industry [11, 20, 58]. Essentially, any static software application security testing (SAST) tool can be converted into a target selection method, with the potentially faulty lines as target locations. A fuzzer can then be used to create a proof of concept, which static analysis usually cannot do on its own. For our analysis, we choose two SAST tools, Rats [29] and Cppcheck [50]. Initially developed by Secure Software Inc. in 2001, Rats is one of the earliest freely available SAST tools and although its development was discontinued in 2013, it stayed relevant as a common academic baseline [21, 38, 47]. Cppcheck is a mature open-source SAST tool with a very active community continuously developing and extending the software. Its availability in package repositories of many major Linux distributions, precise error messages, integration with many popular IDEs and its ease of use makes it a popular choice for initial security assessments of C/C++ code [1, 50]. Second, we want to capture the broader trends in other areas of code assessment, which rapidly turn to learning-based approaches. In particular, we select three source code based vulnerability prediction models for this purpose: ReVeal [14], which is based on graph neural networks, Linevul [30], resorting to a encoder-only transformer based architecture, and a fine-tuned version of the transformer based language model CodeT5+ [71].

Finally, we include one target selection that uses a random scoring of code locations as a baseline for the other models. In the following, we provide a brief overview of our selected methods.

CodeT5+. Based on the transformer architecture, CodeT5+ [71] is a large language model for programming tasks. It is pre-trained on a subset of the CodeSearchNet [35] and GitHub Code datasets[1]. Pre-training consists of a first stage with unimodal tasks such as denoising, followed by a second stage using text-code bimodal data, aiming to improve code generation and understanding tasks. For our experiments, we use the 220 million parameter version and extend it to perform a vulnerability prediction task similarly to Chen et al. [18]. To this end, we complement the model with a prediction head for binary classification as described in [36] and fine-tune it on the DiverseVul dataset [18], consisting of ca. 350k (19k vulnerable) annotated methods representing 150 CWEs.

Cppcheck. The Cppcheck SAST tool is able to perform a variety of checks on C and C++ source code, scanning for undefined behavior and dangerous code patterns that might indicate vulnerabilities [50]. Checks include analysis passes for automatic variable checking, array bounds checking or dead code elimination. A recent study evaluating the tool on Mozilla Firefox found that Cppcheck was able to find 83.5% of the vulnerabilities with only 7.2% false positives [22]. Cppcheck can be used with different pre-built configurations. We use the default setup which includes all of them.

Rats. The *Rough Auditing Tool for Security* (Rats) [29] is one of the earlier rule-based SAST solutions. The tool offers language support for C/C++, Perl, PHP, Python and Ruby with varying degrees of maturity; while it is able to scan C code for more complex bug classes such as Time-of-Check-Time-of-Use, analysis capabilities in Python are constrained to checking for potentially dangerous built-in or library function calls. The results of Rats can be filtered by three types of severity, which we include all. Besides that, we use the default configuration.

Leopard-C. Du et al. propose Leopard as a framework for identifying potentially vulnerable code sites in C/C++ programs [24]. The framework primarily works in two stages, the first one employing code complexity metrics, hereinafter referred to as *Leopard-C*, to sort functions into different bins, and the second one using vulnerability metrics (*Leopard-V*) to rank functions inside those bins. Leopard-C takes into account the cyclomatic complexity of a function, i.e., the number of linearly independent Control Flow paths, and a set of loop metrics: the number of loops and nested loops as well as the maximum nesting level. The intuition is that more complex functions are harder to analyze and reason about.

Leopard-V. With Leopard-V, the authors derive a new set of metrics from common vulnerability causes [24]. More precisely, they select eleven code metrics that aim to capture the following characteristics: the dependency to other functions, the use of pointers, and features of employed control structures. For the first characteristic, they count the number of parameters of the function and the number of parameters to its callees. To measure the use of pointers, they, for example, count the number of variables involved in pointer arithmetic. Lastly, they measure the features of the control structures such as the number of nesting levels or the number of conditional statements without an alternative. Finally, Du et al. sum up all individual metrics into a single vulnerability metric.

[1] https://huggingface.co/datasets/codeparrot/github-code

Sanitizer instrumentation. Our sanitizer based selection methods builds on the idea of ParmeSan [83], SAVIOR [19], and FishFuzz [79], which utilize information about the code regions augmented with instrumentation by the employed sanitizers. This is based on the rationale that sanitizer instrumentation is added at code locations where defective behavior might occur (e.g., array accesses). To implement this, we count the number of sanitizer callbacks added to each function. These callbacks are added during compilation, for example, when memory is allocated or freed, or other memory access that may potentially allow for errors are detected. Consequently, a function with many sanitizer callbacks indicates a higher likelihood for a sanitizer to detect an error in this function than in functions with very few or none added callbacks.

ReVeal. ReVeal [14] is a Graph Neural Network (GNN) for vulnerability detection based on Zhou et al.'s approach of learning program semantics [80]. It uses Code Property Graphs, a holistic code representation combining Abstract Syntax Trees with Dataflow- and Control Flow Graphs [75], as an input to the eight-step GNN and augment it with a classification layer to separate the learning of code representations from the learning of vulnerability indicators. The model is trained on a large real-world dataset of vulnerabilities in the Chromium and Debian Kernel sources. We use the implementation from [67].

Linevul. Using Microsoft's CodeBERT [28], a 125M parameter encoder-only transformer for programming tasks, as a foundation model, Linevul [30] provides line-level vulnerability prediction for C/C++ code. Following results from [36], we use the original model trained on the BigVul dataset [27] but apply normalization to any code input before inference in order to reduce confounding introduced by code style variations.

Recently modified code ("Recent"). Several directed fuzzing approaches resort to recently modified code locations as their targets [7, 12, 55, 81]. Zhu and Böhme [81] even find that most (four out of five) bugs found by OSS-Fuzz are introduced by recent code changes. Therefore, we also include such a target selection in our experiments. To this end, we first assign each function the time it was last modified. We identify the changes based on information from the version tracking system employed by the individual projects. When every function has been assigned a timestamp, we rank them with the most recently changed functions ranking highest.

Random. To provide a baseline for the previous methods, we also include a target selection which outputs a random ranking over all code locations.

5.2 Results and Insights

As discussed in Section 4.1, the problem of finding suitable targets in a project to guide a fuzzer towards can be framed as an information retrieval problem. As such, we approach the evaluation of the target selection methods with a standard metric to measure the quality of retrieval algorithms with, the NDCG.

In Section 4.2, we described the assembly of our dataset based on information from OSS-Fuzz. By reproducing crashes of projects in defective versions, we gathered stack traces which we used to assign a relevance score to each function in the respective project. This way we ended up with a corpus of projects in specific versions

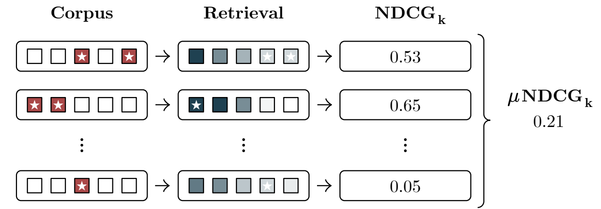


Figure 7: Retrieval and evaluation process. Our crash corpus consists of various projects with functions labelled as relevant (☆) or irrelevant. A target selection method is used to create a ranking of the functions with the k highest ranks forming the retrieval. For each retrieval we calculate the NDCG and average the results across all crashes in every project of our corpus, leading to the μNDCG_k as the retrieval score for a selection method on k sized retrievals.

and functions labelled as relevant or irrelevant, based on whether they appeared in the stack trace of the crash, or not. This is shown in Figure 7. We understand target selection as the retrieval of these relevant functions for each project in a defective version. To that end, the functions are ranked by using a target selection method with the k highest ranking function forming the retrieval. We calculate the NDCG for each of these retrievals and take the mean of the NDCG score, μNDCG_k, to quantify the retrieval performance of the target selection method.

As our labeling is based on crash stack traces and not every function in a given stack trace is defective, we have to deal with a certain degree of label noise, which we counteract by using one metric that under-estimates and one that over-estimates the true quality of our selection methods, the NDCG^- and the NDCG^+, respectively, as described in Section 4.1.

In this section, we compare the retrieval performance of the target selection methods. As noted in Section 4.2, different crash types do occur with different frequencies and are, hence, included in our dataset with varying amounts. The same holds for crashes found by the three sanitizers that we took into account for the crash reproduction. This makes it necessary to take a closer look at the target selection methods with regard to their quality for those categories in our dataset. However, before examining these specific cases, we begin by taking the general retrieval performance into the focus.

General performance. The performance of the target selection methods is shown in Figure 8. To get an intuition of how to interpret the μNDCG_k scores, recall that the NDCG expresses the degree to which the ranking created by a selection method matches an ideal ranking. An ideal ranking would exclusively assign functions from a crash stack trace to the first ranks; thus, an ideal NDCG score of 1.0 would express that a given target selection algorithm is able to match this ranking and thus predict crashing functions perfectly.

One particularity we observe in the case of the under-estimating NDCG^- is the drop in retrieval performance for for $k > 1$ retrieved functions for Leopard-V, as can be seen in Figure 8 a). When considering solely the top-ranking function (i.e., $k = 1$), the Leopard-V method exhibits a NDCG score of 0.13. This implies that the initial function in the ranking of this method aligns with that of an ideal

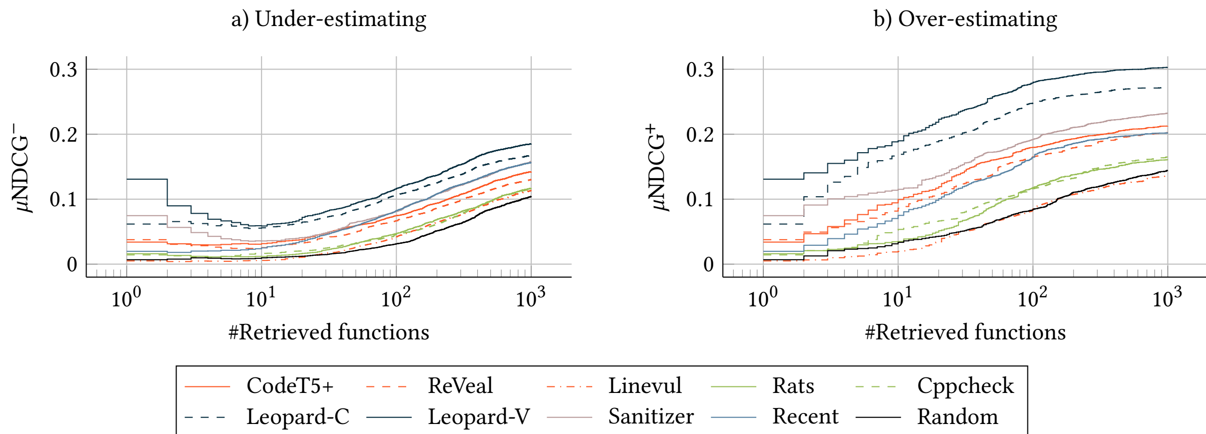


Figure 8: Mean retrieval scores of the target selection methods. Mean NDCG scores of the target selection methods across the whole corpus for the first k retrieved functions. Target selection methods belonging to the same general class of approaches, i.e., SAST tools, vulnerability prediction models, code metrics, and sanitizer instrumentation-based share the same color.

algorithm in 13% of instances. Put differently, the highest ranking function of Leopard-V is part of a crash's stack trace in 13% of the cases across every crash in our corpus. With an increasing number of retrieved functions, i.e., greater values for k in the plot, the under-estimating NDCG$^-$ drops, which shows that the target selection method does not continue to rank the other functions from the stack trace as highly. When taking into account that the under-estimating matching policy assigns an equal relevance to every function from the stack trace, this indicates, that only one of the stack trace's functions is considered very likely to be defective by the selection method. This could be explained by the fact that in many cases only one function from the stack trace actually is defective in practice. This behavior does not arise in the over-estimating case, as in this case only the highest ranking function from the stack trace is assigned a relevance $r > 0$.

In order to further investigate the relationship of the respective methods to each other, we conducted a Mann-Whitney-U test to identify significance[2]. Furthermore, we refer to a method as *dominant* over another, if its distribution of NDCG$_k$ scores is significantly higher for every number k of retrieved functions.

Let us briefly summarize our insights in this regard. First, we shift our focus to the case in which at least 25 functions were retrieved ($k \geq 25$) as this marks the point from which on every method, except for Linevul, significantly outperforms the random baseline. In this case, we find four clusters of target selection methods which can be recognized particularly well in the over-estimating plot in Figure 8 b). The clusters are characterized by the fact that the methods within a cluster do not dominate one another, but each method of one cluster dominates each method of the one beneath. The first cluster is formed by Leopard-V and Leopard-C which outperform every other method. The second one consists of the sanitizer-based method, CodeT5+, ReVeal, and the recent code changes heuristic. This is followed by the SAST tools cluster containing Rats and Cppcheck which still dominate the last cluster, Linevul and the random scoring. The original Linevul model likely performs similarly to the random baseline as it picks up spurious correlations with code formatting [36].

When we shift our view to less than 25 retrieved functions ($k < 25$), we find a different picture. We observe that the sanitizer-based method performs better than the other three methods with which it forms a cluster for $k \geq 25$ for a few number of returned functions. More precisely, it significantly outperforms the next best methods CodeT5+ and ReVeal for $k < 5$. Also, for the same interval, the method based on how recently a code location was last modified performs significantly worse than the next best methods CodeT5+ and ReVeal. This shows how methods that do not significantly differ for a larger number of retrieved functions can perform vastly different when queried for a few number of relevant functions.

To understand the practical implications of these results, we take a closer look at the case of AFLChurn [81]. In their case study the authors compare their proposed directed fuzzer, AFLChurn, which can operate on targets from a continuous target selection method, to AFLGo, which requires a discrete one. For the target selection they assign each code locations a continuous score based on how recently it was modified. Consequently, AFLGo, requiring a discrete set of targets, must decide on a threshold score or a maximum amount of code locations to select as targets for the fuzzing process. When measuring the time to expose a crash in libhtb, one of their target programs, the authors find that AFLGo takes 3 hours and 12 minutes on average while AFLChurn clocks in at 2 hours and 1 minute. Now, our experiments add a new angle to explain this performance gain. We observe that a target selection based on how recently a code locations was last modified only picks up performance with an increasing number of retrieved functions compared to the sanitizer or even code metric-based methods. Consequently, in this case a fuzzer that requires a discrete selection of targets and must therefore decide on a threshold score or maximum amount of targets cannot profit as well as one operating with a continuous target selection method can.

[2]We consider $p < 0.05$ statistical significant. We publish p-values together with our source code.

We hypothesize that the difference of the approaches would have been less pronounced for a method that also performs well for a low number of retrieved functions, such as Leopard-V, a sanitizer-based method or Leopard-C. Also, we conjecture that a target selection approach should always be chosen in accordance with the requirements of the fuzzing approach. For example, machine learning-based approaches, such as our fine-tuned CodeT5+ or ReVeal, might work well for fuzzers that can operate on continuous target selections, while showing sub-optimal performance for those that cannot. However, more research is necessary in this direction to better substantiate these hypotheses.

> Target selection methods based on software metrics perform significantly better than every other considered method. The best software metric, Leopard-V, correctly captures as much as 13% of the crashes with its highest ranking function across the whole corpus of more than 1600 crashes. This makes it the most natural and really only viable candidate for fuzzing approaches which require a discrete selection method.

Impact of sanitizers. Our corpus consists of twice as many bugs found by address sanitizer than memory sanitizer or undefined behavior sanitizer combined. Therefore, only taking the retrieval performance on the whole corpus into account largely reflects the performance to retrieve functions from stack traces of crashes found by address sanitizer. This potentially conceals good performance measures with regard to crashes of one of the less frequent sanitizers. The individual performances of the target selection methods broken down by sanitizers are shown in Appendix B. They further support the dominance of Leopard's methods across most numbers of retrieved functions. However, especially for the crashes discovered by the memory sanitizer, the sanitizer-based target selection method works almost as well.

To further inspect the variance in retrieval performance for an individual selection method, we exemplary show the retrieval quality of Leopard-V for the three sanitizers in Figure 9a), indicating that the impact of which sanitizer was used to find a crash on the retrieval performance is limited.

Impact of crash types. Just like with the sanitizers, we can conduct a finer grained assessment of the selection methods performance by breaking them down with respect to the crash types. The resulting retrieval performances are depicted in Appendix C. We observe that the Leopard methods, again, perform best across all examined crash types with the gap to the next best method, the sanitizer-based method, being most pronounced for heap buffer overflows and less pronounced for segmentation faults.

To inspect how strong the retrieval performance depends on the crash type, we take a look at the quality of the method performing strong across all crash types (i.e., Leopard-V) in greater detail. We depict the retrieval performance for every crash type of which we have at least 50 samples in the dataset in Figure 9b). The color strength indicates how prevalent the respective crash type is in our corpus. It shows that even the best performing selection method's retrieval quality varies greatly across the various crash types. Hence, unlike with the sanitizers used to discover the crashes which only have limited impact on the retrieval performance, the quality greatly depends on the crash type. Some crash types, such as heap buffer overflows can be detected very well using the Leopard-V method while others, such as stack buffer overflows, can hardly be detected.

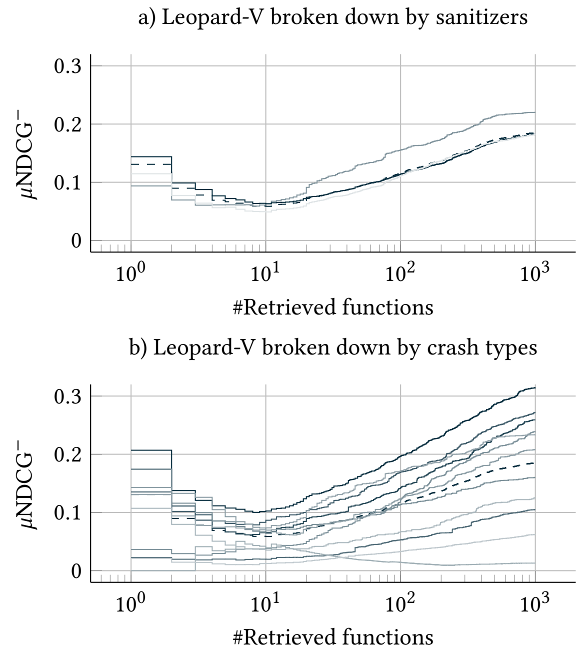


Figure 9: Retrieval performance of Leopard-V across different sanitizer and crash types. Each solid line represents a crash type or sanitizer, respectively. The color strength reflects the prevalence the respective sanitizer and crash type in the dataset. The performance across the whole dataset is added in dashed, for reference.

> Software metric-based target selection methods perform significantly better than any other method across most types of crashes and sanitizers. The only other method close to their performance in some cases is the sanitizer-based one.

6 LIMITATIONS

In our experiments, we investigate the effectiveness of target selection methods in identifying worthwhile targets for directed fuzzers. For this, we treat target selection as an information retrieval problem and compile a dataset from the source code and tracebacks of various crashes we reproduced from the OSS-Fuzz project.

False positives. Underlying our analysis is the assumption that a target which is part of a crash's traceback would also have constituted a valuable target for a directed fuzzer. However, there is a semantic gap between a target's occurrence in a traceback and its value for a directed fuzzer, i.e., how well it aids in triggering the root cause of a crash. It is possible that in some scenarios the

root cause of the crash is not part of the traceback, for example, in case of intricate data-dependencies which have to be met or for multi-threaded or multi-process applications. As we resort to the traceback for our label source, this could lead to false positive labels in our dataset. Yet, previous work has shown that using the traceback of crashes as targets for a directed fuzzer does in fact reduce the time to expose the crashes compared to an undirected fuzzing approach [7, 15, 25]. This indicates that the traceback is indeed a good approximation of the crash's root cause for this application.

False negatives. As the ground truth for our evaluation, we resort to crashes found by the OSS-Fuzz project. This poses the risk of missing crashes that have not yet been identified by the project. In particular, individual targets could be erroneously assigned as not being involved in any crash, when in fact the respective crash has just not yet been found by the fuzzers (i.e., a false negative). While this could potentially lead to an underestimation of the actual scoring quality of the target selection methods, the fact that OSS-Fuzz continuously fuzzes each project with >200 cores per project on average[3] alleviates this risk.

Target granularity. In our experiments, we focus on function level target granularity. While some selection methods and our ground-truth information from the tracebacks would have allowed for a more fine-grained evaluation, for example on line level, this would have excluded several of the other methods. We thus adopted a meet-in-the-middle approach on function level which allows for all target selection methods to be compared to one other while still offering a fine enough granularity to be useful for directed fuzzing. This is shown by the fact that most publications from our literature analysis in Section 3, which introduce a new target selection method, use a function level granularity [13, 24, 42, 77, 79].

7 RELATED WORK

Our systematization of target selection methods for directed fuzzers is related to two types of prior works: surveys of fuzzing literature and analyses aimed at enhancing specific components of the fuzzing pipeline.

Literature surveys. Several surveys on fuzzing research have been conducted throughout the years. Although they are methodologically similar to each other and our work, their respective focus varies. First, there are surveys that take the whole fuzzing pipeline into account. Manès et al. [48] and Liang et al. [44], for example, both introduce general multi-step models of the fuzzing process and survey existing literature with regard to each step in their model. More closely related to our work is the survey by Wang et al. [68] who focus on directed fuzzers. To that end, they identify several characteristics of directed fuzzers for which they examine prior works. While most of these characteristics are concerned with how the fuzzer operates, one characteristic covers the method by which its targets are selected. However, as their focus is on the fuzzer itself rather than on its preceding target selection method, they merely identify which method was used but do not examine it any further.

[3] as of March 2021

Other fuzzing surveys take a more specific perspective and focus on how certain methods are applied in the fuzzing pipeline or challenges that arise when applying fuzzing to particular application areas. That is, for example, how machine learning techniques are used for fuzzing [57, 69] or the application of fuzzing to find flaws in embedded devices [26, 52], respectively.

Lastly, surveys such as those by Schloegel et al. [60], Klees et al. [41] or Kim et al. [40] take a meta perspective and study fuzzing research itself. To that end, they examine the process conducted to evaluate fuzzers in various publications. Based on their findings they can derive information about the general validity of the research field as well as recommendations on how to conduct an evaluation ideally.

Enhancing fuzzer components. In addition to surveying publications on directed fuzzing, we also focus on systematically investigating the step preceding directed fuzzers; namely, the methods employed to select their targets. This is related to prior works which have conducted experiments on individual steps of the fuzzing pipeline. Böhme et al. [9], for example, study various power- and search-strategies to improve the seed scheduling part of a fuzzer, Wu et al. [73] compare different setups for a mutation strategy, and Herrera et al. [33] focus on the seed selection and compare several different methods for that purpose. In contrast, our work focuses on the target selection, which has not yet been studied in-depth, and is, thus, orthogonal to other improvements.

8 CONCLUSION

Our analysis adds a new piece to the puzzle of understanding fuzzing performance. Previous research in directed fuzzing focused on faster and deeper exploration of targets. While this challenge continues to hold enough problems for future work, we demonstrate that the targets within the programs also play a crucial role and the effectiveness of directed fuzzing stands or falls with the quality of the targeted code locations. Our work thus helps to decouple orthogonal factors of performance and provides a better understanding of the *where to fuzz* in the discovery of defects.

Furthermore, the results of our analysis provide a surprising insight: Despite extensive research on locating interesting code for fuzzing, classic software metrics still outperform more sophisticated counterparts in target selection. Given the capabilities of some pattern-based methods, this is unexpected and suggests significant room for improvement. In particular, we consider recent machine learning models a promising alternative, as these models can acquire remarkable skills and thus potentially push the performance of target selection forward, generating a better basis for directed fuzzing in practice.

We make our source code and data publicly available online at https://github.com/wsbrg/crashminer.

9 ACKNOWLEDGEMENTS

This work was funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - (390781972), the European Research Council (ERC) under the consolidator grant MALFOY (101043410), and the German Federal Ministry of Education and Research under the grant BIFOLD24B.

REFERENCES

[1] Andrei Arusoaie, Stefan Ciobâca, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. 2017. A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 161–168. https://doi.org/10.1109/SYNASC.2017.00035

[2] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring Deep State Spaces via Fuzzing.. In *Proc. of the IEEE Symposium on Security and Privacy*. 1597–1612. https://doi.org/10.1109/SP40000.2020.00117

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[4] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. 2023. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge.. In *Proc. of the USENIX Security Symposium*.

[5] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing.. In *Proc. of the USENIX Security Symposium*. 1985–2002.

[6] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation.. In *Proc. of the USENIX Security Symposium*. 235–252.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2329–2344.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing.. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2329–2344. https://doi.org/10.1145/3133956.3134020

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain.. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 1032–1043. https://doi.org/10.1145/2976749.2978428

[10] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking.. In *Proc. of the International Conference on Software Engineering*. 1621–1633. https://doi.org/10.1145/3510003.3510230

[11] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. *Moving Fast with Software Verification*. Technical Report. Facebook Inc. (Meta).

[12] Sadullah Canakci, Nikolay Matyunin, Kalman Graffi, Ajay Joshi, and Manuel Egele. 2022. Targetfuzz: Using darts to guide directed greybox fuzzers. In *Proceedings of the 2022 ACM on Asia conference on computer and communications security*. 561–573.

[13] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, Jiajia Li, and Tao Wei. 2023. ODDFuzz: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing.. In *Proc. of the IEEE Symposium on Security and Privacy*. 2726–2743. https://doi.org/10.1109/SP46215.2023.10179377

[14] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2022).

[15] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer.. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2095–2108. https://doi.org/10.1145/3243734.3243849

[16] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[17] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search.. In *Proc. of the IEEE Symposium on Security and Privacy*. 711–725. https://doi.org/10.1109/SP.2018.00046

[18] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 654–668.

[19] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing.. In *Proc. of the IEEE Symposium on Security and Privacy*. 1580–1596. https://doi.org/10.1109/SP40000.2020.00002

[20] Timothy Clem and Patrick Thomson. 2021. Static Analysis at GitHub: An Experience Report. *Queue* 19, 4 (sep 2021), 42–67. https://doi.org/10.1145/3487019.3487022

[21] Roland Croft, Dominic Newlands, Ziyu Chen, and M. Ali Babar. 2021. An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '21)*. Association for Computing Machinery, New York, NY, USA, Article 8, 12 pages. https://doi.org/10.1145/3475716.3475781

[22] José D'Abruzzo Pereira and Marco Vieira. 2020. On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects. In *2020 16th European Dependable Computing Conference (EDCC)*. https://doi.org/10.1109/EDCC51268.2020.00025

[23] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating Vulnerability Assessment through PoC Migration.. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 3300–3317. https://doi.org/10.1145/3460120.3484594

[24] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: identifying vulnerable code for vulnerability assessment through program metrics.. In *Proc. of the International Conference on Software Engineering*. 60–71. https://doi.org/10.1109/ICSE.2019.00024

[25] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. Windranger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks.. In *Proc. of the International Conference on Software Engineering*. 2440–2451. https://doi.org/10.1145/3510003.3510197

[26] Max Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. 2022. Embedded fuzzing: a review of challenges, tools, and solutions. *Cybersecurity* 5, 1 (2022), 18.

[27] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)* (Seoul, Republic of Korea). Association for Computing Machinery, New York, NY, USA, 508–512. https://doi.org/10.1145/3379597.3387501

[28] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[29] Fortify. 2013. RATS on Google Code. https://code.google.com/archive/p/rough-auditing-tool-for-security/issues.

[30] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *International Conference on Mining Software Repositories (MSR)*.

[31] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing.. In *Proc. of the IEEE Symposium on Security and Privacy*. 679–696. https://doi.org/10.1109/SP.2018.00040

[32] István Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations.. In *Proc. of the USENIX Security Symposium*. 49–64.

[33] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 230–243.

[34] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning.. In *Proc. of the IEEE Symposium on Security and Privacy*. 36–50. https://doi.org/10.1109/SP46214.2022.9833751

[35] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436

[36] Erik Imgrund, Tom Ganz, Martin Härterich, Lukas Pirch, Niklas Risse, and Konrad Rieck. 2023. Broken Promises: Measuring Confounding Effects in Learning-based Vulnerability Discovery. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, Maura Pintor, Xinyun Chen, and Florian Tramèr (Eds.). ACM, 149–160. https://doi.org/10.1145/3605764.3623915

[37] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection.. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[38] Arvinder Kaur and Ruchikaa Nayyar. 2020. A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. *Procedia Computer Science* 171 (2020), 2023–2029. https://doi.org/10.1016/j.procs.2020.04.217 Third International Conference on Computing and Network Communications (CoCoNet'19).

[39] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-box Fuzzing guided by Data Dependency.. In *Proc. of the USENIX Security Symposium*.

[40] Tae Eun Kim, Jaeseung Choi, Seongjae Im, Kihong Heo, and Sang Kil Cha. 2024. Evaluating Directed Fuzzers: Are We Heading in the Right Direction?. In *Proceedings of the 32st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[41] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing.. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2123–2138. https://doi.org/10.1145/3243734.3243804

[42] Johannes Krupp, Ilya Grishchenko, and Christian Rossow. 2022. AmpFuzz: Fuzzing for Amplification DDoS Vulnerabilities.. In *Proc. of the USENIX Security Symposium*. 1043–1060.

[43] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing.. In *Proc. of the USENIX Security Symposium*. 3559–3576.

[44] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[45] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. 2022. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs.. In *Proc. of the IEEE Symposium on Security and Privacy*. 2078–2095. https://doi.org/10.1109/SP46214.2022.9833683

[46] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration.. In *Proc. of the IEEE Symposium on Security and Privacy*. 2693–2707. https://doi.org/10.1109/SP46215.2023.10179296

[47] Rahma Mahmood and Qusay H. Mahmoud. 2018. Evaluation of Static Analysis Tools for Finding Vulnerabilities in Java and C/C++ Source Code. *CoRR* abs/1805.09040 (2018). arXiv:1805.09040 http://arxiv.org/abs/1805.09040

[48] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331.

[49] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: high-coverage testing of software patches. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, (ESEC/FSE)*.

[50] Daniel Marjamäki. 2007. Cppcheck: Tool for static C/C++ code analysis. https://github.com/danmar/cppcheck.

[51] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2022. Linear-time Temporal Logic guided Greybox Fuzzing.. In *Proc. of the International Conference on Software Engineering*. 1343–1355. https://doi.org/10.1145/3510003.3510082

[52] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices.. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[53] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level directed fuzzing for {use-after-free} vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 47–62.

[54] Lin Padgham, Young Lee, Shazia Sadiq, Michael Winikoff, Alan Fekete, Stephen MacDonell, Dali Kaafar, and Stefanie Zollmann. [n. d.]. CORE Rankings. https://www.core.edu.au/conference-portal

[55] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 2019. 1dvul: Discovering 1-day vulnerabilities through binary patches. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 605–616.

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

[57] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. 2019. A Review of Machine Learning Applications in Fuzzing. arXiv:1906.11133 [cs.CR]

[58] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Communications of the ACM (CACM)* 61 Issue 4 (2018), 58–66. https://dl.acm.org/citation.cfm?id=3188720

[59] Nico Schiller, Merlin Chlosta, Moritz Schloegel, Nils Bars, Thorsten Eisenhofer, Tobias Scharnowski, Felix Domke, Lea Schönherr, and Thorsten Holz. 2023. Drone Security and the Mysterious Case of DJI's DroneID.. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[60] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 137–137.

[61] Lukas Seidel, Dominik Christian Maier, and Marius Muench. 2023. Forming Faster Firmware Fuzzers.. In *Proc. of the USENIX Security Symposium*.

[62] Kostya Serebryany. 2017. {OSS-Fuzz}-Google's continuous fuzzing service for open source software. (2017).

[63] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. 2022. MC2: Rigorous and Efficient Directed Greybox Fuzzing.. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2595–2609. https://doi.org/10.1145/3548606.3560648

[64] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing.. In *Proc. of the IEEE Symposium on Security and Privacy*. 803–817. https://doi.org/10.1109/SP.2019.00052

[65] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis.. In *Proc. of the IEEE Symposium on Security and Privacy*. 2194–2211. https://doi.org/10.1109/SP46214.2022.9833761

[66] M. Sutton, A. Greene, and P. Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

[67] VulDetProject. 2021. VulDetProject Implementation of ReVeal. https://github.com/VulDetProject/ReVeal.

[68] Pengfei Wang, Xu Zhou, Tai Yue, Peihong Lin, Yingying Liu, and Kai Lu. [n. d.]. The progress, challenges, and perspectives of directed greybox fuzzing. *Softw. Test. Verification Reliab.* 34, 2 ([n. d.]).

[69] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. 2020. A systematic review of fuzzing based on machine learning techniques. *PLOS ONE* 15, 8 (2020), e0237749.

[70] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[71] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv preprint* (2023).

[72] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, and Tie-Yan Liu. 2013. Conference on Learning Theory (COLT).

[73] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All.. In *Proc. of the International Conference on Software Engineering*. 1634–1645. https://doi.org/10.1145/3510003.3510174

[74] Valentin Wüstholz and Maria Christakis. 2020. Targeted greybox fuzzing with static lookahead analysis.. In *Proc. of the International Conference on Software Engineering*. 789–800. https://doi.org/10.1145/3377811.3380388

[75] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs.. In *Proc. of the IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[76] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. 2023. DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing.. In *Proc. of the USENIX Security Symposium*.

[77] Lei Zhang, Keke Lian, Haoyu Xiao, Zhibo Zhang, Peng Liu, Yuan Zhang, Min Yang, and Haixin Duan. 2022. Exploit the Last Straw That Breaks Android Systems.. In *Proc. of the IEEE Symposium on Security and Privacy*. 2230–2247. https://doi.org/10.1109/SP46214.2022.9833563

[78] Yujian Zhang, Yaokun Liu, Jinyu Xu, and Yanhao Wang. 2023. Predecessor-aware Directed Greybox Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 40–40.

[79] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets.. In *Proc. of the USENIX Security Symposium*.

[80] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks.. In *Proc. of the Conference on Neural Information Processing Systems (NeurIPS)*. 10197–10207.

[81] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing.. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 2169–2182. https://doi.org/10.1145/3460120.3484596

[82] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning.. In *Proc. of the USENIX Security Symposium*. 2255–2269.

[83] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing.. In *Proc. of the USENIX Security Symposium*. 2289–2306.

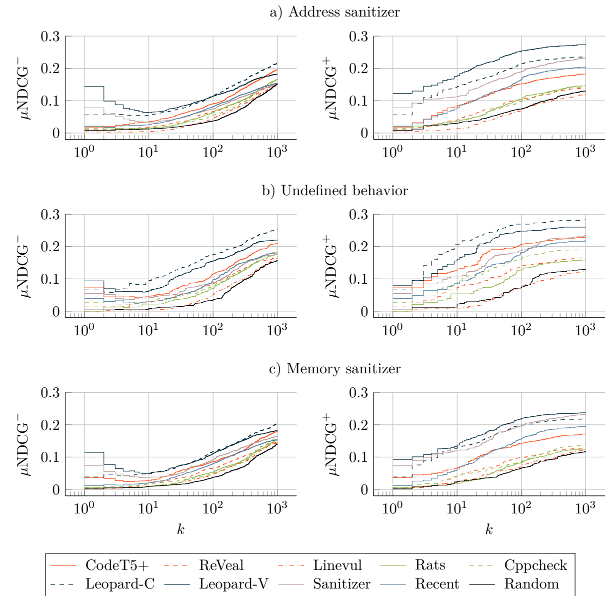
A ADDITIONAL EXPERIMENTS

Alongside the impact of the sanitizer and crash type on the performance of the target selection methods, we also analyzed how the traceback lengths of the crashes might influence the results.

While measuring the traceback lengths' impact is seemingly straightforward, it comes with it own hurdles. In particular, some projects, such as libraw and libpcap, differ greatly in their crashes' average traceback length of 5.2 and 12.4, respectively. The same holds for crash types such as the use of an uninitialized value and a stack overflow whose traceback lengths differs by 230 functions on average. This has to be taken into account when evaluating the impact of the traceback length's. Otherwise the performance on individual projects or crash types would indirectly be measured in this assessment.

To account for this, we focus on the mean percent increase/decrease between the longest and shortest traceback for every combination of projects and crash types. We find that there is only a negligible change of the results for the under-estimating NDCG (less than 0.1%). For the over-estimating NDCG, the impact is higher, but still only shows a limited impact measuring a 2% performance increase for longer tracebacks. The different behaviour can be explained by the optimistic and pessimistic matching strategies applied for the over- and under-estimating NDCG. While the former rewards if *any* function from the traceback was ranked highly, the latter one expects *every* function from the traceback to be at high positions. Highly ranking any function is significantly simpler for longer tracebacks which could explain the impact of the traceback length.

B RETRIEVAL PERFORMANCE BY SANITIZERS



C RETRIEVAL PERFORMANCE BY CRASH TYPES