SyzRisk: A Change-Pattern-Based Continuous Kernel Regression Fuzzer

Gwangmu Lee
gwangmu.lee@epfl.ch
EPFL

Duo Xu
duo.xu@epfl.ch
EPFL

Solmaz Salimi
s.salimi@sharif.edu
Sharif University of Technology

Byoungyoung Lee
byoungyoung@snu.ac.kr
Seoul National University

Mathias Payer
mathias.payer@nebelwelt.net
EPFL

Abstract

Syzbot continuously fuzzes the full Linux kernel to discover latent bugs. Yet, around 75% of recent kernel bugs are caused by recent patches, dubbed *regression bugs*. Regression fuzzing prioritizes inputs that target recently or frequently patched code. However, this heuristic breaks down in the kernel environment as there are too many patches (and therefore too many targets).

To improve regression fuzzing, we note that certain code change patterns (e.g., modifying GOTO) carry more risk of introducing bugs than others. Leveraging this observation, we introduce SyzRisk, a continuous regression fuzzer for the kernel that stresses bug-prone code changes. SyzRisk introduces *code change patterns* that allow for identifying risky code changes. After systematically estimating the risk of suspected change patterns under various circumstances, SyzRisk assigns more weight to risky change patterns. Using the accumulated corpus from prior continuous fuzzing, SyzRisk further prioritizes mutation inputs based on the observed weights.

We simulated the pattern creation from developers using 146 known Linux kernel root causes including 38 CVE root causes and collected 23 risky change patterns. The evaluation shows that the pattern-based weighting method highlights root-cause commits 3.60x more compared to the heuristic of simply targeting recent and frequent changes. Our evaluation of the Linux kernel v6.0 demonstrates that SyzRisk records a 61% speedup in bug exposure time compared to Syzkaller, while discovering the most complete set of bugs across all compared fuzzers.

CCS Concepts

• **Security and privacy → Operating systems security**.

Keywords

Continuous Fuzzing, Kernel Security, Regression Testing, Development Study, Code Analysis

ASIA CCS '24, July 01–05, 2024, Singapore

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Gwangmu Lee, Duo Xu, Solmaz Salimi, Byoungyoung Lee, and Mathias Payer. 2024. SyzRisk: A Change-Pattern-Based Continuous Kernel Regression Fuzzer. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '24), July 01–05, 2024, Singapore.* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Bugs in an operating system (OS) kernel expose the entire system. Kernel fuzzing is widely used to proactively detect such bugs, limiting their impact. However, the Linux kernel is huge with a high code turnover—the kernel consists of over 26 million lines of code as of July 23, 2023, and 529k lines of code are effectively added every three months. The large size and fast change call for a massive amount of fuzzing time for thorough checking. Researchers attempt to address this issue by making kernel fuzzing more efficient [24, 30, 39, 47], but the most well-adopted method is *continuous fuzzing* (e.g., Syzbot [7]) where the most recent kernel image is indefinitely fuzzed alongside development.

However, while continuous fuzzing inherently rechecks the entire kernel, recent findings suggest that the vast majority of new bugs are attributed to a tiny fraction of code: the code changed by patch commits [8, 60, 64]. Known as *regression bugs*, [64] suggested that around 77% of bugs reported by OSSFuzz [6] are regressions. The situation in the Linux kernel is no different, as the Syzbot record in the last two years (i.e., 2021~2022) suggests that around 75% of newly discovered bugs are attributed to patch commits that were updated less than *one* year before they are first detected. Regression bugs are therefore the dominant source of bugs in the kernel.

Motivated by similar findings, AFLChurn [64] introduced *regression fuzzing* that focuses on recently patched code areas, facilitating the detection of regression bugs through fuzzing. The basic idea is to assign *weights* to the code, prioritizing inputs that incur higher weights during execution. AFLChurn further incorporated a heuristic code weighting method that gives a higher weight to *recent* and *frequent* code modifications, based on the general observation in the software engineering community [23, 35, 36].

However, this heuristic weighting method breaks down under the kernel environment as the amount of *recent* and *frequent* modification itself is huge. In 2022, the Linux kernel accepted an average of 219 commits every day. This amounts to almost 526k lines of changed (i.e., added or deleted) code per month, which is of similar size as a stand-alone user-space project—for reference, ImageMagick 7.1.0-60 [4] consists of 663k lines total, with only 95k lines being

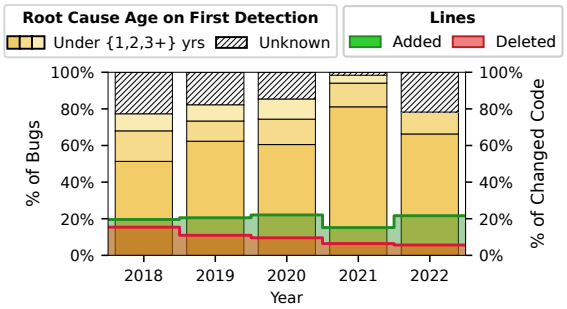


Figure 1: Percentage of bugs by the age of root causes on the first detection (stack graph) and changed lines of code per year (area graph). "Unknown": bugs that Syzbot could not have found the cause commit out of recent commits.

changed every month. The excessive number of patch modifications distracts regression fuzzing to benign modifications (i.e., the modifications that do not cause bugs), making it fail to put due testing time to *root-cause* modifications (i.e., the modification that provides the fundamental cause of bugs).

The existing approach assumes that each code modification is equally bug-prone upon its introduction, and the recentness and frequency are the sole major factors determining the likelihood of bugs. However, we note that not every code modification is equally bug-prone. Certain modifications have more potential side-effects that call for more thorough developer attention, otherwise easily introduce new bugs.

We investigated 146 Linux kernel root-cause modifications in 2020~2021 and established three observations that characterize root-cause modifications. First, some changes are more likely to cause bugs, hence risky; changing a struct field, for example, requires adjustment of *all* use locations or otherwise may cause bugs. Second, the risk of modification—a set of code changes—compounds as it involves multiple risky changes; more risky changes mean more complexities that developers need to handle, some of which are likely to be missed as they grow. Finally, some risky changes are project-specific; modifying `goto`s, for example, is especially risky in the kernel because they are usually coupled with resource cleanup.

Notice that all observations point to the developer's involvement in making a certain code change risky, meaning that they can suspect which changes *may* be risky when they make changes or when retrospectively looking into root causes for bug fixes. Leveraging this, we design SyzRisk, a continuous kernel regression fuzzer that stresses the suspected root-cause code changes. SyzRisk provides kernel developers with a simple interface to compose a set of suspected risky change patterns, which SyzRisk proceeds to estimate their risk under various circumstances (e.g., size of modification or containing subsystem) using known root causes. SyzRisk further matches the patterns with patch modifications and increases the weights as they match more patterns, suggesting a higher chance of root causes. Finally, SyzRisk prioritizes the corpus inherited from the prior continuous fuzzing campaign based on the collected weights during execution.

To demonstrate the effectiveness of SyzRisk, we emulated the pattern creation scenario from developers using the 146 Linux root-cause modifications and composed 23 recurring code change patterns with 31 lines of code per pattern on average. Our evaluation

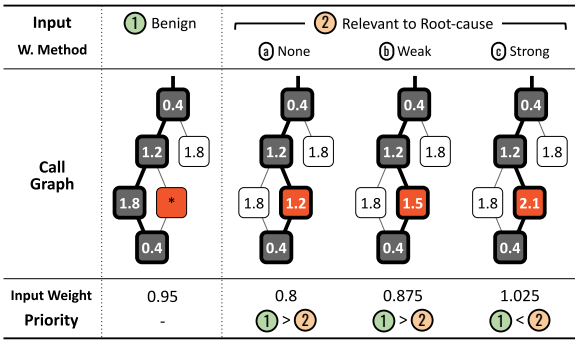


Figure 2: Conceptual illustration of regression fuzzing and the impact of weighting methods. Each box and number represent a function and its weight, and the red box represents the function with a root-cause modification.

of past Linux kernel commits shows that the collected patterns not only remain valid but also highlight the root cause modifications 3.60x more likely than the simple heuristic of just using the latest commits, including root causes that our patterns are not directly derived from. We further compared SyzRisk to state-of-the-art regression fuzzing methods with the Linux kernel v6.0, showing a 61% improved bug discovery time than Syzkaller [22].

In summary, we present the following contributions:

- We present SyzRisk, a change-pattern-based continuous kernel regression fuzzer, which provides a simple interface to describe suspected risky change patterns and automatically estimates their *risks*.
- We propose a systematic way to assess the *risk* of code changes and demonstrate its effectiveness in regression fuzzing to stress unknown root-cause modifications.
- We present 23 risky change patterns in the Linux kernel that highlight root causes 3.60x better than the heuristic weighting method, which allows for a 61% improved bug discovery time against Syzkaller.
- We open-source SyzRisk to facilitate pattern development and future research on kernel regression fuzzing: `https://github.com/HexHive/SyzRisk`

2 Background and Motivation

SyzRisk builds on and extends continuous kernel fuzzing and regression fuzzing. To highlight differences and our contributions, we here highlight the techniques and some of their limitations.

2.1 Continuous Kernel Fuzzing

Continuous kernel fuzzing (e.g., Syzbot [7]), as the name suggests, *continuously* fuzzes kernel images along with version development. Specifically, Syzbot maintains multiple fuzzing instances so that they can periodically rebuild the kernel images from designated branches with various kernel configs and sanitizers. When continuing with rebuilt kernel images, Syzbot reuses the amassed corpus from older revisions because most inputs in the corpus remain valid to cover most of the kernel even after revision updates.

Limitation. By design, continuous kernel fuzzing rechecks the entire kernel space every time the kernel is updated, but a majority

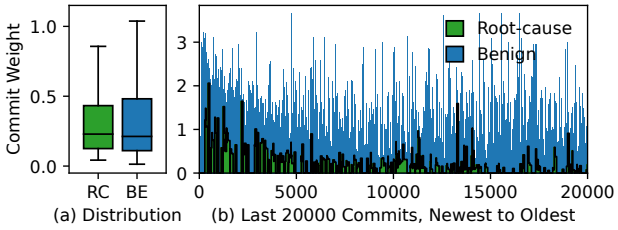


Figure 3: Weight distribution of kernel commits between Jul 1, 2018 and Jul 1, 2019 (the reference date). RC and BE denote root cause and benign commits. In (a), the top, the notch and the bottom of the boxes represent the 25th, 50th and 75th percentile, and the whiskers span the standard 1.5 IQR.

```
--- a/include/linux/kvm_host.h
+++ b/include/linux/kvm_host.h
@@ -432,2 +432,2 @@ struct kvm_memslots {
-  struct kvm_memory_slot memslots[KVM_MEM_SLOTS_NUM];
+  struct kvm_memory_slot memslots[];
 };

@@ -568,2 +568,2 @@ struct kvm_memslots *kvm_alloc_memslots()
   for (i = 0; i < KVM_MEM_SLOTS_NUM; i++)
-  slots->id_to_index[i] = slots->memslots[i].id = -1;
+  slots->id_to_index[i] = -1;

@@ 1032 @@ search_memslots(struct kvm_memslots *sls)
 struct kvm_memory_slot *memslots = sls->memslots;
 // no pointer check for 'memslots'.
 if (gfn >= memslots[slot].base_gfn &&
```

Figure 4: Root cause of CVE-2020-36313 (use-after-free).

of kernel bugs are caused by patch changes—known as *regression bugs*. To confirm how many kernel bugs are regressions, we investigated all Syzbot-reported bugs[1] in 2018~2022 that Syzbot could have successfully performed the root cause bisection. Figure 1 shows the percentage of bugs that Syzbot successfully identified the root-cause commits, confirming that 86% of bugs have identifiable cause commits. We further classified the bugs by the age of their root cause commits on the first detection, which revealed that nearly 75% of bugs were caused by *0-year-old* commits in the last two years (i.e., 2021~2022), meaning that they were younger than one year when the bugs were discovered.

Orthogonally, we note that only a small portion of the kernel code is changed every year. Figure 1 also shows the percentage of added and deleted lines compared to the total lines of code at the beginning of the year. On average, only 20% and 9% of source code are added or deleted per year. This suggests that even though new bugs in the kernel are dominated by regression bugs, continuous kernel fuzzing spends a large chunk of fuzzing time in unchanged code that is unlikely to exhibit bugs.

2.2 Regression Fuzzing

Motivated by the prevalence of regression bugs, AFLChurn [64] proposed *regression fuzzing* to facilitate regression bug discovery in user-space projects. Basically, regression fuzzing puts *weights* on modifications so as to decide which inputs should be prioritized over others. Figure 2 illustrates how regression fuzzing utilizes weights. In this example, input 1 visits the functions weighted by $\{0.4, 1.2, 1.8, 0.4\}$, and input 2a visits $\{0.4, 1.2, 1.2, 0.4\}$. Regression fuzzing calculates the weight of inputs by averaging the visited weights, which corresponds to 0.95 for input 1 and 0.8 for input 2a. The fuzzer then prioritizes input 1 by giving it a higher chance of selection or a longer mutation time.

AFLChurn further introduced a heuristic weighting method that gives higher weights to *recent* and *frequent* modifications, which also coincides with the intuition that they must be the most relevant to regression bugs. Specifically, given t as the elapsed days since the modification on a given function and f as the total number of modifications on it until the reference date, AFLChurn defined the weight of the function as $1/t \times \log(f)$ to incentivize recent and frequent modifications.

Limitation. However, these heuristics break down in the kernel environment as there are too many *recent* and *frequent* modifications. As mentioned in Section 2.1, the Linux kernel has at most 29% of changed code every year. While this portion is small compared to the entire kernel, this still amounts to around 500k~600k lines of changed (e.g., added or deleted) code every month, which alone is comparable to a middle-to-large user-space project in size.

This results in the root-cause modifications, which should be focused on by regression fuzzing, lost in an overabundance of benign modifications. To confirm this, we checked if the heuristics can distinguish root-cause modifications properly by comparing the weights of benign commits with root cause commits.[2] Figure 3 shows the weight distribution of root cause (RC) and benign (BE) commits between Jul 1, 2018 and Jul 1, 2019, where the weighting formula was adjusted for kernel fuzzing so as to reflect its low throughput compared to user-space fuzzing ($10/(t+9) \times \log(f)$). The distribution shows that root-cause commits barely stand out from benign commits, having only 8% higher weights than median benign commits.

Implication. The heuristic of selecting recent patches only weakly targets true root cause modifications. A large amount of modifications weakens the signal and regression fuzzing de-prioritizes them, compromising the efficiency of regression discovery. Figure 2 illustrates three weighting methods, each of which differs in its highlighting power. Suppose we have two inputs where input 1 only visits benign functions while input 2 visits the root-cause function. While regression fuzzing is supposed to prioritize input 2, the weak weighting method (b) cannot prioritize it because input 1 has an even higher weight than input 2, preventing regression fuzzing from thoroughly testing the root-cause function.

This can be avoided by introducing a stronger weighting method, which effectively reduces the chance of such occasions. For example, as the strong weighting method (c) in Figure 2 raises the weight of the root-cause function significantly, it can tolerate a "noise weight" from the benign function in input 1 (i.e., $1.8 < 2.1$) and prioritize input 2. This suggests that *a weighting method should highlight root-cause modifications significantly* to prioritize relevant inputs.

[1] Except 124 erroneous reports that confused the same compile error (in `elf.c`) as reproduced crashes.

[2] Commit weights are calculated as the average of contained functions.

```
1  --- a/net/rds/connection.c
2  +++ b/net/rds/connection.c
3  @@ -193,6 +193,10 @@ __rds_conn_create()
4    conn = kmem_cache_zalloc(rds_conn_slab);
5    conn->c_path = kcalloc(npaths, sizeof(...));
6    ...
7  - if (is_outgoing && trans->t_prefer_loopback) {
8  -   trans = &rds_loop_transport;
9  + if (trans->t_prefer_loopback) {
10 +   if (unlikely(is_outgoing)) {
11 +     kmem_cache_free(rds_conn_slab, conn);
12 +     conn = ERR_PTR(-EOPNOTSUPP);
13 +     goto out;
14 +   }
15   }
```

Figure 5: Root cause of CVE-2021-45480 (memory-leak).

3 Characterizing Root-cause Modifications

As elaborated in Section 2.2, effective regression fuzzing boils down to highlighting potential root-cause modifications over benign ones. To gain insight into the differentiating characteristics of root- ause modifications, we investigated 146 Linux bugs including 38 CVEs in 2020 and 2021. In this section, we present three key observations that we noticed during our investigation.

1) Some changes are more likely to cause bugs (i.e., risky).

While any code changes can potentially cause problems, some changes are more *likely* to cause bugs, hence *risky*. This is because different code changes create a different amount of side-effect that developers need to address. This amount grows as the code change involves more global effects or more intricate logic, rendering developers likelier to miss some side-effect.

For example, Figure 4 shows the root-cause commit of CVE-2020-36313 that changes a struct field `memslots` from a static array to a dynamic array. As it is changing the semantics of the struct field, it potentially needs to adjust all use locations to prevent them from incorrectly assuming stale semantics. This commit indeed adjusts the initialization function for `struct kvm_memslots` (Line 10-11), but it fails to address one use location in `search_memslots()` that implicitly assumes `memslots` as a static array, which subsequently exhibits use-after-free.

2) The risk compounds as it involves more risky changes.

Not only some changes are riskier than others, but the risk of a modification—a group of contiguous changes—also gets amplified as it consists of more risky changes. For example, Figure 5 shows the root cause of CVE-2021-45480, which attempted to add a new exception in the network initialization function `__rds_conn_create()`. The modification involves several risky changes, including;

- Managing memory; `kmem_cache_free()` in Line 11.
- Changing initialization; `conn` in Line 12, allocated in Line 4.
- Changing error code logic; `-EOPNOTSUPP` in Line 12.
- Handling a `goto` control flow; `goto` in Line 13.

While the developer was dealing with the complexities of all the risky changes, they missed to cleanup `conn->c_path` (Line 5) as a part of the new exception, causing a memory leak. The rationale here is that each risky change presents additional complexities that a developer has to address, so the overall modification gets more likely to be problematic as risky changes accumulate.

3) Some risky change patterns are project-specific.

While the risk of some changes comes from the language itself, others come from how the project conventionally uses the language construct. For example, while the complexities in Figure 4 come from the global nature of structure definitions, the last two risky changes in Figure 5 comes from how kernel developers use a set of return codes (i.e., to maintain inter-functional semantics) and the `goto` construct (i.e., to handle exceptional control flow, including resource cleanup).

4 Formalizing Risk of Code Change

The observations in Section 3 suggest that the risk of modifications indicates how likely a given modification will cause problems, hence being a root cause. In this section, we formalize the risk of modifications to highlight potential root causes.

Risk of individual code change. Considering that a risk is supposed to represent how likely a code change is involved in problems, we can formulate the risk of a code change as follows. Suppose the probability of a change c appearing in root-cause modifications is $P_{bug}(c)$, and the probability in benign modifications is $P_{benign}(c)$. Then the risk of the change $R(c)$ can be described as the probability $P_{bug}(c)$ relative to the probability $P_{benign}(c)$;

$$R(c) = \frac{P_{bug}(c)}{P_{benign}(c)}. \tag{1}$$

In this formulation, a change can be considered *risky* if the risk of the change $R(c)$ is greater than 1.

Risk of modification. Similar to the risk of a single change, the risk of a modification—a group of contiguous changes—can be described as the relative probability of root-cause and benign modifications featuring *all* constituent changes simultaneously. In other words, the risk of a certain modification m is described as;

$$R(m) = \frac{P_{bug}(\wedge_{n=1}^{N} c_n)}{P_{benign}(\wedge_{n=1}^{N} c_n)}, \tag{2}$$

where $\{c_1, ..., c_N\}$ is a set of all changes in the modification m. This can be further approximated as the *product* of each change's risk (Equation 1), as the probability of each change appearing in any modifications is approximately independent of each other unless one pattern does not imply (or require) another.[3]

$$R(m) \sim \frac{P_{bug}(c_1) \times \cdots \times P_{bug}(c_N)}{P_{benign}(c_1) \times \cdots \times P_{benign}(c_N)} = \prod_{n=1}^{N} R(c_n), \tag{3}$$

Notice that the formalized risk of a modification also gets amplified by the risk of constituting code changes as noted in Observation 2.

5 SyzRisk

Directed by the observations in Section 3 and using the formalized code change risk in Section 4, we present SyzRisk, the change-pattern-based continuous kernel regression fuzzer that shifts weights to potential root cause modifications. SyzRisk allows kernel developers to describe suspected risky change patterns through a simple interface and automatically estimates the risks of the supplied patterns. SyzRisk then increases fuzzing weights for matching code

[3] For example, the code changes in struct fields do not *require* changing GOTOs.

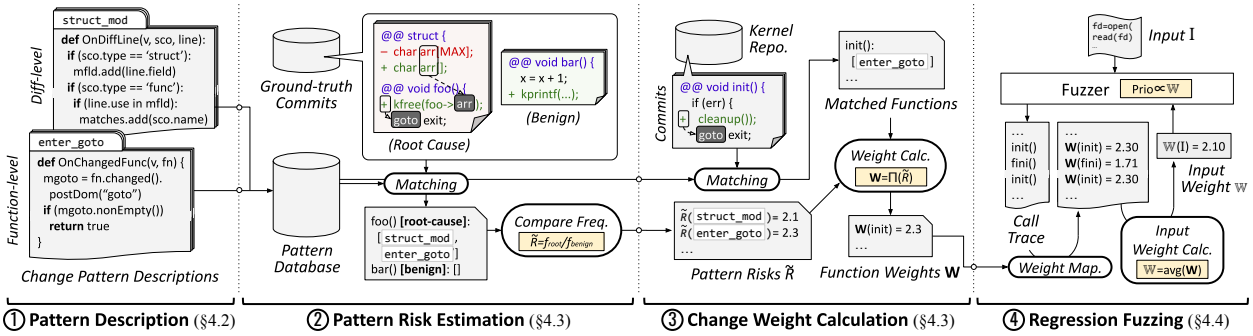


Figure 6: Overall workflow of SyzRisk.

Callback	Parameter/Return	Called at...
`OnCommitBegin (sha, date)`	`sha`: Hexsha of the commit. `date`: Commit date.	Beginning of commit.
`OnDiffLine (v, sco, line)`	`v`: Version (added or deleted). `sco`: Enclosing scope information. `line`: Changed line information.	Each changed line.
`OnCommitEnd()`	`RET`: Set of matched functions.	End of commit.

Table 1: List of callbacks for a diff-level description.

Callback/API	Parameter/Return	Called at...
`GetReqAnalysis()`	`RET`: Set of required analysis.	Before matching.
`OnChangedFunc (v, fn)`	`v`: Version (before or after). `fn`: Changed function body.	Each changed function.
`GetMetadata (meta)`	`meta`: Type of metadata. `RET`: Requested metadata.	(API)

Table 2: List of callbacks and an API for a function-level description.

changes (i.e., more likely to be root causes), adjusting weights over time based on their age. Figure 6 shows the workflow of SyzRisk.

5.1 Workflow and Usage Model

Pattern description. Developers first describe code change patterns that they suspect risky or bug-prone whenever they spot them (Section 5.2). Developers may notice potentially risky changes at various stages of development; when their code changes are causing too many follow-up modifications in different locations (e.g., modifying struct fields) or when certain code changes keep appearing in root causes while they are fixing crashes.

Pattern risk estimation. Given change patterns, SyzRisk proceeds to estimate the actual risk of each pattern (Section 5.3). To do so, SyzRisk leverages a ground-truth dataset that consists of all root-cause and benign commits until the current point. SyzRisk matches the patterns to both kinds of commits and estimates the risk of each pattern by comparing the relative match frequency of a given pattern in root-cause and benign commits (Observation 1). At this stage, developers may also polish the quality of their patterns iteratively (e.g., for higher risks) using the estimated risks.

Code change weight calculation. With estimated pattern risks, SyzRisk calculates the weight of all code changes in the repository by matching the patterns to them. SyzRisk amplifies the weight of code changes by the risks of their matched patterns, as it suggests a higher likelihood of causing crashes (Observation 2).

Regression fuzzing. Finally, the SyzRisk fuzzer uses the calculated code weights to prioritize input mutation (Section 5.4). SyzRisk makes use of the previous continuous fuzzing campaign by inheriting its input corpus, which already covers most kernel parts but needs to be prioritized based on the relevance to risky code changes. SyzRisk decides such relevance per input based on the code weights collected during its execution.

5.2 Pattern Description and Matching

Reflecting on Observation 1 and 3, SyzRisk allows developers to describe suspected risky change patterns by implementing predefined callback functions. Writing a description is analogous to writing a small plugin for the LLVM framework [31] (e.g., AST visitor or IR pass), except that the amount of required implementation is much smaller (i.e., tens of lines).

SyzRisk provides two ways of describing change patterns, depending on the subject of analysis; diff-level and function-level. A diff-level description analyzes the diff of a commit, deducing patterns based on the changed (i.e., added or deleted) lines in the diff. A function-level description analyzes the body of a changed function—the function that contains at least one changed line—deducing a pattern from high-level program graphs within the function (e.g., AST, CFG or DDG). Appendix A describes the step-by-step matching procedure with the exact timing of callbacks.

5.2.1 Diff-level Description Table 1 shows a list of callback functions for a diff-level pattern description. The detailed explanations of the callbacks are as follows.

OnCommitBegin(sha, date). SyzRisk calls this callback before it begins matching a new commit. Developers may utilize this callback to initialize the internal states or identify the current commit using `sha` and `date`, the SHA hash and date of this commit.

OnDiffLine(v, sco, line). SyzRisk calls this callback when it encounters a changed line (i.e., added or deleted; indicated by `v`) while scanning through the diff of the current commit. By implementing this callback, developers can construct a pattern-matching state machine using the provisioned information by SyzRisk as follows. `sco` provides the information on the enclosing scope such as `type` and `name`. Currently, SyzRisk recognizes five types of enclosing scopes; `function`, `struct`, `enum`, `initializer` and `global`. `line` represents the changed line, and developers can either use pre-analyzed

token information (e.g., `field` and `use`) or manually analyze code using regular expression.

OnCommitEnd(). SyzRisk calls this callback when the current commit is ended. This callback is expected to return a set of matched functions in the current commit.

5.2.2 Function-level Description Table 2 shows a list of callbacks and an API for a function-level pattern description. The detailed explanations of the callbacks and the API are as follows.

GetReqAnalysis(). SyzRisk calls this callback before starting a matching process, where it is expected to return a set of required analyses for this pattern description. Currently, SyzRisk supports three intra-functional analysis—AST, CFG and DDG—as supported by the back-end pattern matcher [5].

OnChangedFunction(v, fn). SyzRisk calls this callback for each changed function body. This callback is effectively called twice with two versions before and after the commit, indicated by `v`. `fn` is the body of the changed function with the analysis result and tagged changed lines, on which developers can make Joern [5] queries. This callback is expected to return whether or not the function is matched to the pattern.

GetMetadata(meta). To compensate the limitation of intra-functional analysis, SyzRisk provides this API so that developers can leverage meta-functional information not available within a function body. The supported types of metadata are; i) the SHA hash of the enclosing commit, with which developers can get the results on a commit level, and ii) a list of function attributes (e.g., `__init` or `__release`), from which developers can assume the intended purpose or behavior of the function.

5.3 Risk Estimation and Weight Calculation

With the matching results, SyzRisk decides the weights of the changed functions by reflecting on the risk of matched patterns. Specifically, SyzRisk first estimates the *risks* of the supplied patterns by examining how frequently they appear in known root cause modifications compared to benign modifications. SyzRisk then calculates the weights of each changed function based on the risk of matched patterns.

5.3.1 Pattern Risk Estimation As formulated in Section 4, the risk of a change pattern $R(c)$ can be described as $P_{bug}(c)/P_{benign}(c)$, where $P_{bug}(c)$ and $P_{benign}(c)$ is the probability of a change c appearing in root-cause and benign modifications. As directly measuring these probabilities is impossible, SyzRisk rather approximates them by examining how frequently the pattern appears in ground-truth dataset—consisting of known root cause and benign modifications.

Furthermore, SyzRisk uses different subsets of ground truth to estimate different types of risk under various circumstantial factors. Formally, let X be a circumstantial factor such as the commit size or the changed subsystem. If $f_{bug}(c|X)$ and $f_{benign}(c|X)$ are the frequencies of a change pattern c in root-cause and benign modifications *that satisfy the circumstantial factor* X, the estimated risk of the pattern $\tilde{R}(c|X)$ *if the enclosing modification satisfies the circumstantial factor* X is;

$$\tilde{R}(c|X) = \frac{f_{bug}(c|X)}{f_{benign}(c|X)}. \tag{4}$$

Currently, SyzRisk considers three types of risks depending on the considered circumstantial factors as follows;

- **General Risk $\tilde{R}(c|\phi)$.** This risk is the closest to the general description of a risk, where the circumstantial factor ϕ is satisfied by *any* modification.
- **Commit-Size Risk $\tilde{R}(c|C_{sc})$.** This risk reflects the correlation between a change pattern and the size (i.e., the number of changed functions) of the commit that it belongs to. To be specific, let $sc = \lfloor \log_2(s+1) \rfloor$ be the size class of the commit size s. Then, the circumstantial factor C_{sc} is satisfied by the modifications in the commit of the size class sc.
- **Subsystem Risk $\tilde{R}(c|S_{sys})$.** This risk reflects the correlation between a change pattern and the changed subsystem. Formally, if sys represents a subsystem, the circumstantial factor S_{sys} is satisfied if a modification changes the subsystem sys.

Notice that the estimated risks can be updated incrementally as kernel development continues because it only requires matching newly identified root cause and benign modifications and adding them to the frequency counts. SyzRisk currently bounds the estimated risk within the range of $[0.5, 10]$ to avoid under-estimating or over-estimating risks due to the lack of ground truth. The upper bound was chosen by a larger factor than the lower bound (i.e., 10x higher vs. x2 lower than 1) to favor potentially risky matches even if the high risk may be transient due to the limited dataset.

5.3.2 Function Weight Calculation After estimating the risk of patterns, SyzRisk calculates the weights of each matched function. Conceptually, the weight can be defined as proportional to the risk of modifications inside the function (i.e., $R(m)$ in Equation 2), but the ground truth dataset would never be sufficient to estimate every combination of change patterns. Instead, we again approximate the risk of modifications using Equation 3 by multiplying all the risks of the matched patterns. This approximated risk should be upper-bounded, however, because patterns may develop correlations between each other as a modification gets heavy. For example, both changing error code logic and `goto` control flows are likely to appear simultaneously in heavy modifications such as creating an entire function. This can break the assumption that patterns are independent and result in over-estimating heavy modifications.

One additional factor we need to incorporate here is the elapsed days t since the modification has been made so that we can fade out old, hence well-tested modifications. To incorporate this, we additionally multiply a time-dependent exponential factor to the estimated risk. Formally, if $\{c_1, ..., c_N\}$ is a set of matched change patterns inside a function F, the weight of the function $\mathbf{W}(F)$ is;

$$\mathbf{W}(F) = \text{bound}(\prod_{n=0}^{N} \tilde{R}(c_i)) \times 2^{-t/T}. \tag{5}$$

Here, $\tilde{R}(c)$ is the maximum estimated risk under all circumstantial factors considered, and $\text{bound}(x) = -Be^{-x/B} + B$ puts a soft upper-bound on the approximated risk at B, which currently SyzRisk sets it to the eighth power of the average general risk (i.e., around 256). T is a fade-out constant that reduces function weights by half every T days after their modification date. If a function ends up with multiple weights (e.g., multiple commits have modified it), SyzRisk simply takes the maximum weight.

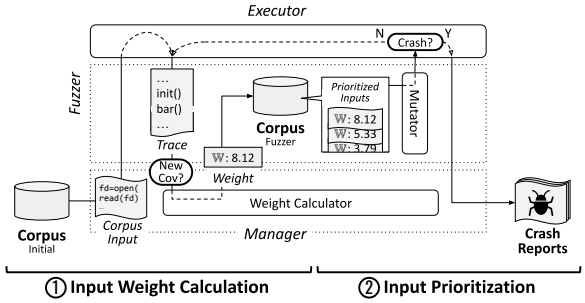


Figure 7: Design of SyzRisk runtime components.

5.3.3 Ground-truth Collection While accurately estimating risks requires a large (generalizable), sound (no false-negative) and complete (no false-positive) ground-truth dataset, in practice, collecting such a dataset is notoriously difficult as it is intertwined with identifying root-cause modifications [9, 18, 53, 55, 56]. The following elaborates on how SyzRisk addresses these issues.

Ensuring a large dataset. While it is generally difficult to identify the root cause *modifications*, we can rather easily identify root-cause *commits* in the Linux kernel, thanks to the commit convention. To be more specific, the commit convention encourages kernel developers to tag the commit (`Fixes:`) that their commits are supposed to fix, and especially for *crash*[4] fixes, the commit that introduces them—hence root causes. Based on this observation, SyzRisk first seeks the commits with `Fixes` tags to identify fix commits and check whether such commits are fixing crashes using Syzbot reports and relevant keywords in the commit message. Appendix D elaborates on fix commit identification. SyzRisk then traces back to the tagged commits to collect root-cause commits. In reverse, SyzRisk collects any untagged commits for benign samples.

Ensuring soundness. Clearly, the dataset cannot include the root-cause commits that have not yet been identified, creating false negatives in the root-cause dataset and false *positives* in the benign dataset in reverse. However, the false-negative root causes are eventually covered as the kernel development continues and developers keep tagging root-cause commits. Moreover, we note that the soundness issue is only about risk estimation; all root-cause commits get weighted with the estimated risks regardless of whether they are included in the dataset.

Ensuring completeness. Even if we can identify root-cause commits, some modifications in such commits might be *benign*. Specifically, root-cause commits contain increasingly more benign modifications as they get larger, because not all modifications contribute to crashes. On the other hand, we also cannot exclude all large root-cause commits either, as some root-cause modifications might span across multiple functions. To bypass this issue, SyzRisk limits the size of the commit to up to seven changed functions unless the risk estimation requires larger commits (e.g., commit-size risk in Section 5.3.1) and assumes all modifications are root causes.

5.4 Regression Fuzzing

With the calculated function weights, SyzRisk performs regression fuzzing by prioritizing inputs that visit more weighted functions, hence bearing higher relevance to root-cause modifications. Figure 7 illustrates the design of the SyzRisk runtime components. SyzRisk first calculates the weight of an input as the average weight of functions that it visited during execution, then prioritizes high-weight inputs by giving them higher chances for mutation.

Input weight calculation. SyzRisk calculates the weight of an input as the average weight of all visited functions during its execution. Formally, let $\mathcal{F}$ be the list of visited functions during the execution of the input I. Then, the weight of the input $\mathbb{W}(I)$ is;

$$\mathbb{W}(I) = \text{avg}(\mathbf{W}(F), \forall F \in \mathcal{F}). \tag{6}$$

At the beginning of fuzzing, SyzRisk-manager feeds every input in the initial corpus from the prior continuous fuzzing to SyzRisk-executor and calculates the weights of each input, and it continues to calculate the weight of newly discovered inputs during fuzzing in a similar way.

Input prioritization. Given the calculated input weights, SyzRisk-fuzzer prioritizes the high-weight inputs by selecting them with higher probabilities for mutation. Similar to AFLChurn [64], SyzRisk-fuzzer calculates the normalized input weight ω against the maximum input weight in the corpus and decides the likelihood of input selection by multiplying $2^{5(2\omega-1)}$ to the original selection likelihood determined by the base fuzzer (i.e., Syzkaller). SyzRisk does not require simulated-annealing-like priority regulation unlike the original regression fuzzing, as the corpus from the prior continuous fuzzing is presumed to have sufficiently *explored* inputs.

6 Change Patterns

To demonstrate the feasibility of the usage model (Section 5.1), we simulated the pattern description scenario and collected 23 risky change patterns using the 146 root-cause modifications in Section 3. Specifically, we presumed developers creating patterns while analyzing root causes for bug fixes. Table 3 shows the 23 risky change patterns we collected. In this section, we briefly illustrate how we collected them and present the details of two representative patterns: Modified Struct Field and Entering GOTO.

Ensuring initial pattern set. We first started collecting our initial candidate pattern set by examining the root-cause commits of 38 crashing CVEs in 2020 and 2021 and 65 Syzbot-reported crashes fixed in November 2020 and March 2021. We identified root-cause *modifications* that directly caused crashes by referring to where the corresponding fix commit modified, and noted the recurring changes in the root-cause modifications. As a result, we collected 14 suspected risky code change patterns.

Enhancing pattern set. After dropping some candidate change patterns that were too generic (i.e., introducing new memory references) or too hard to generalize (i.e., changing inside double-loops), we composed the patterns with SyzRisk (Section 5.2) and matched them to all `Fixes`-tagged 1,100 root-cause commits in 2020 to check if the patterns cover them all. Then, we analyzed 43 more unmatched root-cause commits in 2020 to cover *all* root-cause commits (except data-only or macro-changing commits, discussed in Section 9) by tuning the existing patterns to cover corner cases and adding nine more patterns (Exported Function, Extracted Statement, Modified Assembly, New State, Pointer Promotion, Split Function, Changed Error Code, Inside Switch, Switch-ctrled Var. Mod.).

[4]Including KASAN-detectable bugs, concurrency bugs and kernel-panics.

```
1 def OnDiffLine(v, sco, line):
2   if (sco.type == "struct"):
3     mfld.add(line.field)
4   elif (sco.type == "func"):
5     u2f[line.use].add(sco.name)
6
7 def OnCommitEnd():
8   return set().union(*[u2f[fld] for fld in mfld])
```

Figure 8: Simplified description of Modified Struct Field.

Refining pattern set. We further estimated the general risks of the collected patterns to check whether they are all *risky enough*. As a result, we dropped one pattern whose general risk was under 1.2 (New Callback) and split two patterns into multiple ([Pointer Arithmetic, Involving GOTO] to [Pointer Arithmetic, Pointer API, Pointer Offset Manip., Entering GOTO, Inside GOTO]) as they were underestimating the risks of individual patterns by implicitly or-ing them.

Checking the generality of patterns. To ensure the general riskiness of the collected patterns throughout time, we additionally estimated the pattern risks with the commit samples before and after when they are derived from (i.e., 2020~2021). Section 8.2 presents the estimated risks, suggesting that all collected patterns continue to be risky over the years.

Statistics. Most of the patterns required no more than 30 minutes to write descriptions, except a few patterns that required reasoning about how to describe (e.g., Pointer Arithmetic; what should be considered as affecting pointer arithmetic?). Overall, nine diff-level patterns and 14 function-level patterns take 32.5 lines and 29 lines on average, respectively. The longest pattern is Extracted Statement (80 lines) that involves string similarity comparison [2], while the shortest pattern is Memory Mgmt. API (15 lines) that essentially only finds pre-defined memory management function calls (e.g., `"*alloc*"`). Table 6 in Appendix E shows the number of investigated root causes per bug type.

Pattern	Summary	Level
Chained Dereference	Involving chained field dereference.	Diff
Exported Function	Exporting previously-local functions.	Diff
Extracted Statement	Moving statements to another function.	Diff
Modified Assembly	Changing assembly instructions.	Diff
Modified Struct Field	Using a modified struct field.	Diff
New State	Using a new state variable. (e.g., #define)	Diff
Pointer Promotion	Promoting a variable to a pointer.	Diff
Split Function	Splitting a single function.	Diff
Struct Casting	Changing struct-to-struct castings.	Diff
Changed Error Code	Changing error codes or their conditions.	Func
Concurrency API*	Changing concurrency API calls.	Func
Entering GOTO	Changing code leading to GOTO.	Func
Finalization	Changing inside a finalization function.	Func
Global Variable	Changing the usage of global variables.	Func
Initialization	Changing inside an initialization function.	Func
Inside GOTO	Changing code inside GOTO.	Func
Inside Switch	Changing code inside a switch-case.	Func
Locked Context	Changing code inside a locked context.	Func
Memory Mgmt. API*	Changing memory management calls.	Func
Pointer API*	Changing pointer manipulation API calls.	Func
Pointer Arithmetic	Changing pointer arithmetic code.	Func
Pointer Offset Manip.	Changing the value of pointer offsets.	Func
Switch-ctrled Var. Mod.	Changing switch-case-controlled variables.	Func

*: Can be described on both diff-level and function-level.

Table 3: List of 23 risky change patterns.

```
1 def OnChangedFunc(v, fn): Boolean {
2   val gotos = fn.gotos()
3   val postdom = gotos.postDominates()
4   val ctrl = gotos.enclosingCond()
5   return ((gotos ++ postdom ++ ctrl).hasChanged())
6 }
```

Figure 9: Simplified description of Entering GOTO.

6.1 Modified Struct Field

Description. Modified Struct Field is a diff-level pattern that searches for any changed usage of modified structure fields. Figure 8 illustrates the simplified description of the pattern. Given a commit diff, it first identifies a set of modified structure fields (`mfld` in Line 3) and pairs of used structure fields and the functions (`ufld` in Line 5). Then it returns the functions using the fields in `mfld` (Line 8).

Rationale. Since any structure field has its own semantics—e.g., what it does or how it is used—that should be globally agreed upon in any code locations, all relevant functions should be adjusted properly if a structure field is modified, hence changing its semantics. Weighting the functions that are using such fields is to check if they are handling the changed semantics properly and to indirectly visit other relevant functions during a fuzzing run.

6.2 Entering GOTO

Description. Entering GOTO is a function-level pattern that checks if changes are made to any statements that directly lead to them. Figure 9 illustrates the simplified description of the pattern. Given a changed function, it first identifies all `goto` statements in the function (Line 2) and the lines and the condition post-dominated by or enclosing them (Line 3-4). Then, it checks whether such lines are changed (Line 5).

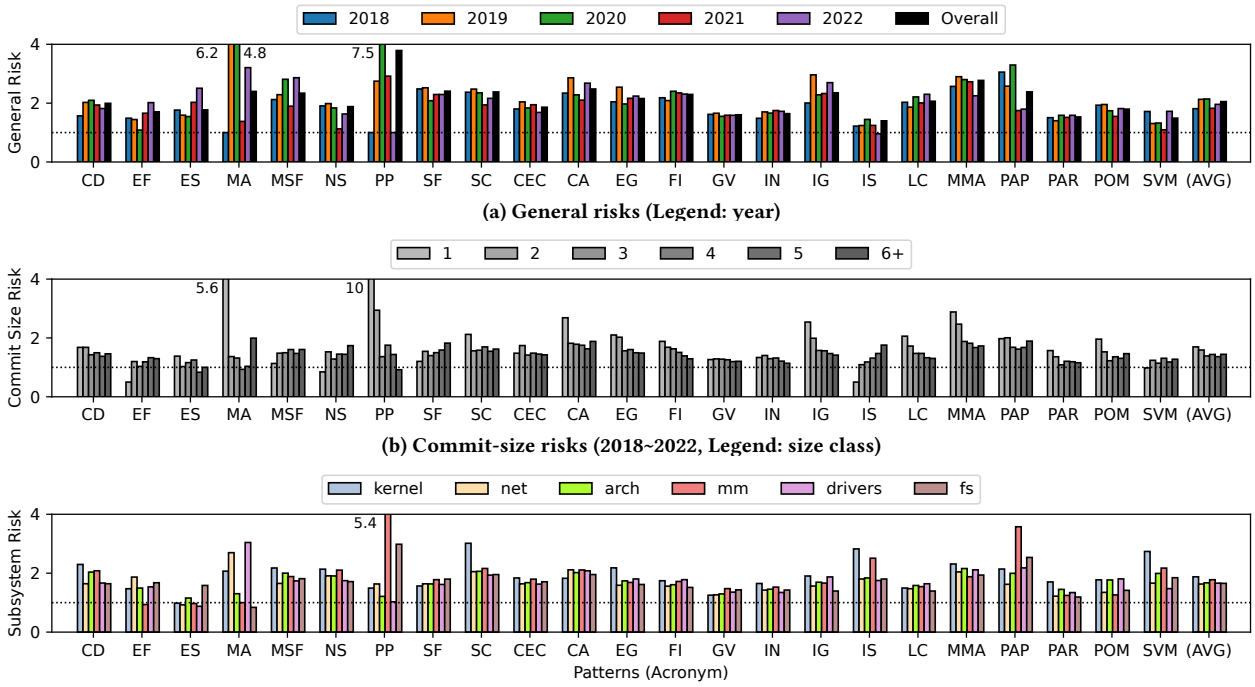
Rationale. As mentioned in Observation 3, the `goto` construct is a prevalent way for kernel developers to handle exceptions, and the lines leading to `goto`s and the conditions that decide their execution directly implement the exceptional logic (e.g., resource cleanups). Weighting such functions is to check if the changed exceptional flows are properly implementing such secondary logic, which can otherwise cause memory bugs (e.g., use-after-free).[5]

7 Implementation

Pre-fuzzing components. For diff-level matching, we implemented SyzRisk in Python with GitPython [3] to scan commits from the repository and extract the body of changed functions. For function-level matching, we incorporated Joern v1.360 [5] for the back-end pattern matcher. The pattern description interfaces (Section 5.2) were written in Python and Scala for diff-level and function-level, respectively. In total, the pre-fuzzing components consist of 4,609 lines of code.

Runtime fuzzer. We based on Syzkaller (revision `2b253ced7f2f` with a mutation bug fix from `6feb842be06b`) aiming at minimizing the changes for the potential merge to Syzkaller. Our prime issue was deriving a list of visited functions from the `kcov` coverage, which is partially done by Syzkaller when generating a crash report. To utilize this, we modified `ReportGenerator` to acquire a list of

[5]The exception logic inside `goto` blocks is covered by Inside GOTO.



(a) General risks (Legend: year)

(b) Commit-size risks (2018~2022, Legend: size class)

(c) Subsystem risks (2018~2022, Legend: subsystem)

Figure 10: Risks of the 23 change patterns. Patterns are in the same order as Table 3. 2022 commits are up to Oct 2, the release date of v6.0. Overall: based on all 2018~2022 commits.

visited functions from non-crashing inputs and `syz-manager` to feed the calculated input weights to `syz-fuzzer`. We additionally modified `syz-fuzzer` to update the input weights and adjust the input selection priority accordingly. In total, we implemented the core regression fuzzing functionality with only 234 lines of code.

8 Evaluation

In this section, we demonstrate the effectiveness of change-pattern-based regression fuzzing by comparing SyzRisk to the state-of-the-art regression fuzzing approaches. To be specific, we present the answers to the following research questions through evaluation.

- **RQ1.** How risky are the collected 23 change patterns in Table 3? (Section 8.2)
- **RQ2.** How much does the change-pattern-based weighting method highlight root-cause modifications, compared to the heuristics on recent and frequent modifications? (Section 8.3)
- **RQ3.** How much does SyzRisk, the change-pattern-based regression fuzzer, outperform other state-of-the-art regression fuzzers in bug discovery time? (Section 8.4)
- **RQ4.** How completely can SyzRisk discover bugs compared to other regression fuzzing methods? (Section 8.5)

8.1 Evaluation Setup

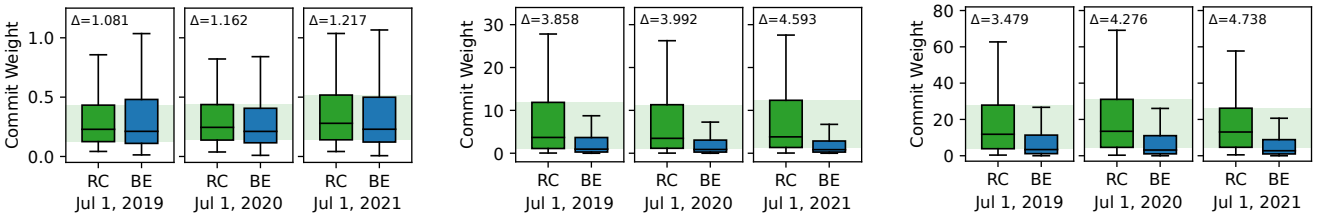
We ran pattern matching, risk estimation and weight calculation on a server node equipped with an Intel Xeon E5-2680 CPU (56 cores) and 256 GB of memory, running Ubuntu 20.04. The pattern matching for one-year-worth kernel commits (approximately 80,000 commits per year) took 3 hours for diff-level patterns and 6 hours for function-level patterns, where function-level pattern matching was done with 40 parallel threads. We note that the pattern-matching process is not only highly parallelizable up to the unit of each individual commit but also a one-time cost as new commits only need to be pattern-matched just once. The risk estimation and the weight calculation combined took less than 20 minutes with five-year-worth matching results.

For the fuzzer evaluation, we utilized four server nodes with an Intel Xeon Gold 5218 CPU (16 cores) with 64 GB of memory, running Ubuntu 20.04. Each fuzzer was set to eight VMs with two executor processors and placed on different nodes so that it does not interfere other fuzzers. While we set up every machine identically, we assigned different server machines to fuzzers every iteration to avoid a possible machine-introduced bias.

8.2 Pattern Risks

To answer **RQ1**, we estimated the risk of the 23 patterns in Table 3 with the ground-truth commits between Jan 1, 2018 and Oct 2, 2022, including 4,742 root-cause commits (indicated by at least one other commit with `Fixes:` tags) out of 404,965 total commits.

General Risk. Figure 10a shows the general risks of the 23 patterns. The colored bars were estimated with the commits of designated years, while the black bars ("Overall") were estimated with all commits from 2018~2022. On average, the general risk of all patterns is 2.05, which means root causes are 2.05x more likely to involve one of them than benign ones. Notice that the evaluated root-cause commits consist of 4,742 commits spanning five years (2018~2022), while the patterns were extracted from only 146 commits spanning



(a) Heuristic weighting method **(b) Pattern-based weighting method ($T = 30$)** **(c) Pattern-based weighting method ($T = 90$)**

Figure 11: Weight distributions of commits within one year from the reference date (Jul 1, {2019,2020,2021}). RC: root-cause commits, BE: benign commits. Δ: ratio between RC and BE medians. The top, the notch and the bottom of the boxes represent the 25th, 50th and 75th percentile, and the whiskers span the standard 1.5 IQR.

two years (2020~2021, Section 6). This suggests that the patterns are well-generalizable to overall root causes in scale and over time.

The three most risky patterns are Pointer Promotion (PP, 3.79), Memory Mgmt. API (MMA, 2.77) and Concurrency API (CA, 2.48), which are generally regarded as dangerous if done carelessly. Both representative patterns in Section 6 show above-par risks, 2.34 for Modified Struct Field (MSF) and 2.16 for Entering GOTO (EG). Some patterns show exceptionally high (over 4) or low (equal to 1) risks in some years due to the lack of commit samples in those years.

Commit-Size Risk. Figure 10b shows the commit-size risks of the 23 patterns per size class ($\lfloor \log_2(size + 1) \rfloor$, Section 5.3.1). As the average suggests, commit-size risks are showing a decreasing trend as the size class increases, while the patterns like Split Function (SF), Modified Struct Field (MSF) and Inside Switch (IS) are showing the opposite trend. Two possible reasons are: i) developers might have become more likely to miss some corner-cases while dealing with more code adjustments (SF and MSF), or ii) rather than being a direct root cause, some changes might have become more likely to appear in root-cause commits as their size grows (IS).

Pointer Promotion (PP), Concurrency API (CA), Inside GOTO (IG) and Memory Mgmt. API (MMA) show an exceptionally high risk in the size class 1 (i.e., commits that modify only a single function). Notice that they are all highly dependent on other functions to work properly—e.g., the modified resource cleanup logic by Inside GOTO requires the outside of the function to recognize the modified logic properly. This means that they are highly risky if the modification is only done locally without addressing other side effects.

Subsystem Risk. Figure 10c shows the subsystem risks of the 23 patterns in six major subsystems. While most patterns show similar risks in every subsystem, some patterns are showing particularly high risks in certain subsystems. For the memory management subsystem (`mm`), Pointer Promotion (PP), Pointer API (PAP) and Inside Switch (IS) patterns are shown to be particularly risky. All these patterns are closely related to the low-level memory management logic, making them particularly risky as they can easily propagate to all in-kernel memory management when handled incorrectly.

For the core kernel subsystem (`kernel`), most patterns regarding the low-level logic handling (e.g., Struct Casting (SC), Inside Switch (IS) and Switch-ctrled Var. Mod. (SVM)) are generally showing high risks. Notice that the switch-related patterns are showing high risks in `kernel` as opposed to the relativly low general and size class risks (Figure 10a and Figure 10b), reflecting that core kernel logic largely depends on delicate case handling. Modified Assembly (MA) shows highly variable risks across different subsystems mainly due to the lack of commit samples, similar to the case in general risks.

8.3 Root Cause Weighting

To answer **RQ2**, we calculated the weight of Linux kernel commits (i.e., the average weight of functions inside) with two weighting methods—heuristic and pattern-based weighting methods—and compared the weight distributions of root-cause and benign commits. For generality, we picked three reference dates for weight calculation (i.e., from Jul 1, {2019,2020,2021}) and considered the commits within one year from each (i.e., to Jul 1, {2018,2019,2020}).

To evaluate a heuristic that focuses on recent commits, we reused AFLChurn's heuristic weighting formula that puts weights on recent and frequent modifications. As in Section 2.2, we adopted the formula more kernel-friendly by stretching the time-dependent factor to reflect the throughput difference between user-level and kernel fuzzing. For the pattern-based weighting method, we used the pattern-based weighting formula (Equation 5) that puts weights on risky modifications. To check how different fade-out constants (T) affect weight distribution, we calculated weights with two different constants; $T = \{30, 90\}$.

Figure 11 shows the distribution of root-cause commit (RC) and benign commit (BE) weights, calculated by different weighting methods. While for heuristics the weights of root cause commits are almost equal benign commits, the pattern-based weighting method significantly raises the weight of root cause commits in both fade-out constants. In particular, more than half of the root-cause commits have higher weights than 75% of benign commits on every reference date, including Jul 1, 2019 whose commits are entirely disjoint from when the patterns are collected. On geomean, the heuristic weighting method raises the median weight of root-cause commits by only 1.15x, while the pattern-based weighting method raises it by 4.14x with $T = 30$ and 4.13x with $T = 90$, which is 3.60x higher than the heuristic weighting method.

The long fade-out constant ($T = 90$) being as good as the short one ($T = 30$) can be understood by the compensation from old root-cause commits. That is, while a long fade-out constant puts weights on more commits that result in the overall increase of commit weights, it also helps highlight more long-lasting (hence old) root-cause commits, keeping the weight of root causes standing out. Appendix B presents two root-cause commit examples showing how typical root-cause modifications get weighted.

Function	Subsys.	Bug Type	Average TTE (mins)				Relative TTE (vs. Syzkaller)		
			Syzkaller	SyzChurn	SR (T=30)	SR (T=90)	Churn	SR (T=30)	SR (T=90)
blkcg_deactivate_policy	block	deadlock	**1,011**	3,721	2,021	1,801	3.68	2.00	1.78
pfkey_send_acquire	net	kernel-panic	1,657	**393**	580	1,010	0.24	0.35	0.61
ext4_bmap	fs	deadlock	1,921	717	**473**	1,447	0.37	0.25	0.75
tty_port_tty_get	drivers	deadlock	2,307	**1,703**	1,802	1,944	0.74	0.78	0.84
jbd2_journal_lock_updates	fs	deadlock	2,552	2,655	**716**	3,212	1.04	0.28	1.26
nilfs_mdt_destroy	fs	use-after-free	**5,886**	8,501	6,232	7,578	1.44	1.06	1.29
si470x_int_in_callback	drivers	use-after-free	6,028	5,833	**1,913**	4,746	0.97	0.32	0.79
sco_conn_del	net	deadlock	7,020	10,080	6,696	**6,220**	1.44	0.95	0.89
ext4_xattr_set_handle	fs	deadlock	7,980	7,681	**5,176**	7,926	0.96	0.65	0.99
ntfs_attr_find	fs	buffer-overflow	8,178	9,151	6,380	**4,858**	1.12	0.78	0.59
vmf_insert_pfn_prot	mm	kernel-panic	8,544	6,979	8,221	**6,898**	0.82	0.96	0.81
detach_extent_buffer_page	fs	null-dereference	9,017	8,277	8,695	**6,460**	0.92	0.96	0.72
evict	fs	deadlock	10,080	3,661	3,675	**2,206**	0.36	0.36	0.22
Geomean							0.86	**0.62**	0.80
Speedup							16%	**61%**	24%

Table 4: Average time-to-exposure (TTE) of 13 bugs discovered by all three iterations, in ascending order of the TTEs from Syzkaller. The bold TTEs are the shortest TTEs among fuzzers. SR: SyzRisk. Timeout: 10,080 minutes (7 days).

8.4 TTE Comparison

To answer **RQ3**, we compared the bug exposure time of SyzRisk to the state-of-the-art regression fuzzing methods with the Linux kernel v6.0. The compared fuzzing methods are as follows;

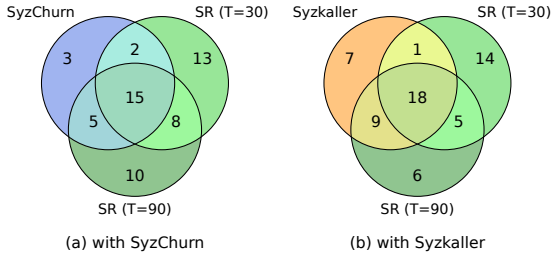
- **Syzkaller.** The baseline vanilla kernel fuzzer, representing a normal continuous fuzzing scenario. We used the same base revision (Section 7) without modification.
- **SyzChurn.** The kernel port of the state-of-the-art user-space regression fuzzer, AFLChurn [64]. We implemented SyzChurn based on SyzRisk with heuristic weights on recent and frequent changes (Section 2.2).
- **SyzRisk.** The Syzkaller-based implementation of the change-pattern-based regression fuzzing. Similar to Section 8.3, we tried two different fade-out constants (Ts) to check their impact; $T = 30$ and $T = 90$.

To emulate the continuous fuzzing scenario, we prepared the initial corpus by running four-VM Syzkaller with the Linux kernel v6.0[6] for seven days on a Intel Core i7-8665U CPU (eight cores) machine with 16 GB of memory. Then with the collected corpus, we ran fuzzers for seven days (10,080 minutes) and repeated it three times to average the time-to-exposure (TTE) of bugs on each fuzzer.

One challenge of averaging TTEs in kernel fuzzing is that many bugs exhibit highly variable TTEs. In our evaluation, more than half of all discovered bugs were missing in at least one iteration. To take the representative bugs that can be reliably measured within the timeout, we only counted the bugs that appear in all three iterations, regardless of which fuzzer discovered them. We also filtered out trivial bugs detected even before mutation. Appendix C elaborates on the high variance of TTEs in kernel fuzzing.

Table 4 shows the average TTEs of discovered bugs. On geomean, SyzRisk achieves the TTE improvement of 61% with $T = 30$ and 24% with $T = 90$, while SyzChurn only improves it by 16%. The high improvement with $T = 30$ is the result of intensively testing root-cause modifications (as opposed to SyzChurn, which wastes fuzzing time in benign modifications) while limiting the fuzzing space to a small number of modifications (as opposed to $T = 90$, which spreads fuzzing time to old modifications). Meanwhile, $T = 90$ reduces the geomean of the long-TTE bugs—the longest seven bugs in Syzkaller TTE—more than $T = 30$ (0.64 vs. 0.74), suggesting that a longer fade-out constant is more effective for long-lasting bugs.

[6]Strictly, the kernel for the initial corpus collection should be one commit before v6.0, but the effect is negligible because the size of a single commit is insignificant compared to the total size of the kernel.



(a) with SyzChurn (b) with Syzkaller

Figure 12: Number of bugs discovered by each fuzzer at least once in three iterations of TTE comparison (Section 8.4).

8.5 Completeness of Discovered Bugs

To answer **RQ4**, we counted the bugs discovered by each fuzzer at least once in the TTE comparison (Section 8.4). Figure 12 shows the number of discovered bugs, where numbers on intersections are specifying the bugs discovered by all intersected fuzzers. Out of all 62 bugs, both SyzRisk with $T = 30$ and $T = 90$ discovered 61.3% of bugs (38 bugs) separately, while SyzChurn and Syzkaller discovered 40.3% (25 bugs) and 56.5% (35 bugs) of all. Notice that even though SyzRisk focuses on a subset of modifications, both of its configurations found even more bugs than Syzkaller that searches the entire kernel indiscriminately.

The effectiveness of SyzRisk over SyzChurn is clear in Figure 12(a), where both of SyzRisk configurations found three to four times more unique bugs than SyzChurn. Figure 12(b) shows the advantage of $T = 30$ over Syzkaller in testing recent modifications, where it discovered a total of 19 bugs that Syzkaller could not have detected. This result is further compensated by $T = 90$ with six uniquely discovered bugs, which incorporates older root causes in weighting.

In total, the combined SyzRisk missed only 9 bugs, discovering 85.5% of bugs (53 bugs) out of all.

9 Discussion

Integration to continuous fuzzing platform. As continuous fuzzing platforms (e.g., Syzbot [7]) typically run multiple fuzzer instances at the same time, the basic integration method would be running SyzRisk instances along with vanilla Syzkaller instances. While this may constrain the compute resource per individual instance, developers can still expect quicker patch modification testing (with SyzRisk) as well as thorough bug discovery (with Syzkaller). To avoid bias in the corpus after a long regression fuzzing, maintainers can periodically import the corpus from co-running Syzkaller.

False-positive and false-negative matches. For false positives, we note that the composed patterns need not to be strict as the pattern matching of SyzRisk does not aim at *detecting* bugs; SyzRisk rather utilizes it to find modifications that are *likely* buggy, which are later intensively tested by fuzzing. For false negatives, we observed that relaxing pattern description with some assumptions can improve the false-negative rate while still keeping the false-positive rate low. Moreover, SyzRisk tolerates some false-negative matches of individual patterns as SyzRisk incorporates *multiple* patterns.

Possible pattern levels and extension. While currently SyzRisk only supports two description levels—diff-level and function-level—there are more potential description levels to incorporate in the future. For example, a pattern can be described on a module level as a spatial expansion of function-level, or it can span across multiple commits as a temporal expansion of diff-level. Furthermore, the existing function level can also be extended with AST matching techniques [20, 43, 48] to incorporate code change information at an AST-node-level granularity. Exploring risky change patterns on these additional or extended levels is one way to improve SyzRisk.

Data-only and macro-modifying commits. While a majority of root-cause commits directly modify function bodies, a few of them modify them in a less obvious way (e.g., modifying macros) or do not modify them at all (e.g., data-only changes), posing a challenge on which functions should be weighted. One direct measurement is finding every function that references them and assigning weights on it, but it requires additional analysis on the code base. Supporting these root-cause commits is one of our future tasks.

Patterns with inverse risks. One interesting research angle is incorporating patterns with *inverse* risks, or the patterns whose appearance *lowers* the risk of modifications. For example, adding a new null-pointer check could be one of such patterns as it rules out some problematic program states. According to the current formulation (Equation 1), such patterns would exhibit less-than-one risks. They can be used to further regulate the weights of benign modifications, but it can also inversely suppress actual root-cause modifications as they do not necessarily prevent all root causes.

10 Related Work

Kernel Fuzzing. Many researchers have attempted to improve kernel fuzzing in terms of performance and effectiveness. Based on the basic kernel fuzzers [22, 44], continuous kernel fuzzing such as Syzbot [7] circumvents the performance problem by fuzzing them indefinitely. A majority of kernel fuzzing research [15, 16, 21, 24, 39, 45–47] optimizes various fuzzing components such as input selection and synthesis, while other research [28, 30, 41] focuses on a specific kernel component or bug type for effective fuzzing. While none of them directly addresses regression bugs, they are orthogonal to SyzRisk and have no restriction to integrate it.

Directed Patch Testing. Patch modifications have long been suspected to be a main source of bugs [8, 60, 64], which sparked many research on testing patches. Directed fuzzing [10, 11, 13, 34, 52, 54, 57] utilizes the control-flow distance metric to guide a fuzzer toward targets. Differential testing [29, 37, 42, 61] attempts to maximize the patch-induced differences and observe any anomalies in them. However, both target only one or a limited number of locations at a time, so are unsuitable for a plethora of patched code (e.g., in the kernel). Some directed fuzzers [14, 32, 38] focus on suspected root-cause code, but they do not consider code changes for targets, wasting time in old, hence unlikely code locations. Regression fuzzing [64] is the fuzzing technique that particularly gears toward regression bugs by putting weights on patched code locations, but the heuristic weighting method fails to highlight potential kernel root causes, making it ineffective on the kernel.

Vulnerability Detection. Pattern matching [1, 5] is a type of static analysis that discovers bug-causing code patterns in a given code base. In contrast to them, SyzRisk searches for *modification* patterns that are *likely* to cause bugs and utilizes fuzzing to confirm the actual bugginess. Some researches utilize machine learning [12, 33, 49, 51, 62] or statistical methods [19, 27, 40] to detect vulnerabilities. Defect prediction [17, 25, 26, 59] and security patch prediction [48, 50, 63] also utilize machine learning to predict root-cause or security patch commits. While they can still get benefits from SyzRisk's risk estimation and weight calculation (Section 5.3) in the application to regression fuzzing, they commonly suffer from high false-positives and lack comprehensible explanations on how a certain change causes vulnerabilities, let alone excluding any form of insight from the developer side.

11 Conclusion

While kernels are dominated by regression bugs, continuous kernel fuzzing wastes time rechecking unmodified code. Regression fuzzing attempts to mitigate this by weighting modified code, but the heuristics fail to highlight kernel root causes, making it ineffective on the kernel. This paper presents SyzRisk, a change-pattern-based regression fuzzer that weights potential root cause by matching risky change patterns to patch modifications. The evaluation of 23 risky change patterns from the Linux kernel shows that SyzRisk highlights root causes in 2018~2021 3.60x higher than heuristics, enabling 61% improved bug discovery time on the Linux kernel v6.0 against Syzkaller.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2_-186974, and DARPA HR001119S0089-AMP-FP-034.

References

[1] CodeQL. https://codeql.github.com/.
[2] difflib. https://docs.python.org/3/library/difflib.html.
[3] gitpy. https://gitpython.readthedocs.io/en/stable/.
[4] ImageMagick. https://github.com/ImageMagick/ImageMagick.
[5] Joern: open-source code analysis platform for C/C++ based on code property graphs. https://joern.io/.
[6] OSSfuzz. https://github.com/google/oss-fuzz.
[7] Syzbot. https://syzkaller.appspot.com/.
[8] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. How long do vulnerabilities live in the code? a Large-Scale empirical measurement study on FOSS vulnerability lifetimes. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 359–376, 2022.
[9] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. AURORA: Statistical crash analysis for automated root cause explanation. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.
[11] Sadullah Canakci, Nikolay Matyunin, Kalman Graffi, Ajay Joshi, and Manuel Egele. TargetFuzz: Using darts to guide directed greybox fuzzers. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 561–573, New York, NY, USA, 2022. Association for Computing Machinery.
[12] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1456–1468, New York, NY, USA, 2022. Association for Computing Machinery.
[13] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery.
[14] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. SAVIOR: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596, 2020.
[15] Mingi Cho, Dohyeon An, Hoyong Jin, and Taekyoung Kwon. BoKASAN: Binary-only kernel address sanitizer for effective kernel fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.
[16] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NtFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 677–693, 2021.
[17] Agnieszka Ciborowska and Kostadin Damevski. Fast changeset-based bug localization with bert. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 946–957, New York, NY, USA, 2022. Association for Computing Machinery.
[18] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 820–831, New York, NY, USA, 2016. Association for Computing Machinery.
[19] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 60–71. IEEE Press, 2019.
[20] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.
[21] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. ACTOR: Action-Guided kernel fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.
[22] Google. syzkaller - linux syscall fuzzer, 2017. https://github.com/google/syzkaller.
[23] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.
[24] HyungSeok Han and Sang Kil Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2345–2358. ACM, 2017.
[25] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. CC2Vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 518–529, New York, NY, USA, 2020. Association for Computing Machinery.
[26] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: An end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45, 2019.
[27] Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, MetriSec '12, page 7–10, New York, NY, USA, 2012. Association for Computing Machinery.
[28] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, page 0. IEEE, 2018.
[29] Hyungsub Kim, Muslum Ozgur Ozmen, Z. Berkay Celik, Antonio Bianchi, and Dongyan Xu. PatchVerif: Discovering faulty patches in robotic vehicles. In *Proceedings of the 32nd USENIX Conference on Security Symposium [Prepublication]*, SEC'23. USENIX Association, 2023.
[30] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.
[31] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
[32] Yuwei Li, Shouling Ji, Chenyang Lyu, Yuan Chen, Jianhai Chen, Qinchen Gu, Chunming Wu, and Raheem Beyah. V-Fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs. *IEEE Transactions on Cybernetics*, 52(5):3745–3756, 2022.
[33] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
[34] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. GREBE: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.
[35] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 181–190, 2008.
[36] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 309–318, 2010.
[37] Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. HyDiff: Hybrid differential software analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1273–1285. IEEE, 2020.
[38] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.
[39] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.
[40] Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 543–548, 2015.
[41] Hui Peng and Mathias Payer. USBFuzz: A framework for fuzzing usb drivers by device emulation. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.
[42] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632, 2017.
[43] Gordon Plotkin. A note on inductive generalization. *Machine Intelligence*, 1970.
[44] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. KAFL: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 167–182, USA, 2017. USENIX Association.
[45] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary. In *Network and Distributed System Security Symposium (NDSS)*, 2019.
[46] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. HEALER: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 344–358, New York, NY, USA, 2021. Association for Computing Machinery.
[47] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758, 2021.

[48] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. GraphSPD: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2409–2426, 2023.
[49] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. Detecting "0-day" vulnerability: An empirical study of secret security patch in oss. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 485–492, 2019.
[50] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. PatchRNN: A deep learning-based system for security patch identification. In *2021 IEEE Military Communications Conference (MILCOM)*, 2021.
[51] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. VulCNN: An image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2365–2376, New York, NY, USA, 2022. Association for Computing Machinery.
[52] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K Le, Dongliang Mu, and Xinyu Xing. Mitigating security risks in linux with KLAUS: A method for evaluating patch correctness. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, 2023.
[53] Xiaoyuan Xie, Tsong Yueh Chen, and Baowen Xu. Isolating suspiciousness from spectrum-based fault localization techniques. In *2010 10th International Conference on Quality Software*, pages 385–392, 2010.
[54] Jiadong Lu Xin Xiong Zhuang Liu Xin Tan, Yuan Zhang and Min Yang. SyzDirect: Directed greybox fuzzing for linux kernel. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, 2023.
[55] Jian Xu, Zhenyu Zhang, W. K. Chan, T. H. Tse, and Shanping Li. A general noise-reduction framework for fault localization of java programs. *Inf. Softw. Technol.*, 55(5):880–896, may 2013.
[56] Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. Automated bug hunting with data-driven symbolic root cause analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 320–336, New York, NY, USA, 2021. Association for Computing Machinery.
[57] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. SemFuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2139–2154, New York, NY, USA, 2017. Association for Computing Machinery.
[58] M. Zalewski. American fuzzy lop, 2017. http://lcamtuf.coredump.cx/afl/technical_details.txt.
[59] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. Deep just-in-time defect prediction: How far are we? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 427–438, New York, NY, USA, 2021. Association for Computing Machinery.
[60] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V Krishnamurthy, Trent Jaeger, et al. Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel. In *2022 Network and Distributed System Security Symposium*, 2022.
[61] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. D2A: A dataset built for ai-based vulnerability detection methods using differential analysis. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '21, page 111–120. IEEE Press, 2021.
[62] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks.* Curran Associates Inc., Red Hook, NY, USA, 2019.
[63] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. SPI: Automated identification of security patches via commits. *ACM Trans. Softw. Eng. Methodol.*, 2021.
[64] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2169–2182, 2021.

A Pattern Matching Procedure

With patterns supplied by developers, SyzRisk performs a pattern matching on every commit. The step-by-step matching procedure is as follows.

Initialization. Before matching, SyzRisk prepares the kernel repository that contains a list of commits and the pattern database presented by developers. SyzRisk optionally accepts a period of time that limits the period of commit dates to consider, otherwise scans all past commits up to now.

```
1  --- a/include/linux/fs.h
2  +++ b/include/linux/fs.h
3  @@ -1520,6 +1521,9 @@ struct super_block {
4  +  errseq_t s_wb_err;
5
6  --- a/include/linux/pagemap.h
7  +++ b/include/linux/pagemap.h
8  @@ -51,7 +51,10 @@ static void mapping_set_error(...)
9     /* Record in wb_err for checkers */
10 -  filemap_set_wb_err(mapping, error);
11 +  __filemap_set_wb_err(mapping, error);
12 +
13 +  /* Record it in superblock */
14 +  errseq_set(&mapping->host->i_sb->s_wb_err, error);
```

Figure 13: Example of a near-median root-cause modification (kernel-panic, committed on Jun 2, 2020).

Matching diff-level patterns. After initialization, SyzRisk first matches all diff-level patterns commit by commit. At the beginning of a new commit, SyzRisk first calls `OnCommitBegin()` of each diff-level pattern to notify the beginning of a new matching and starts scanning down the diff of the commit to its direct parent. SyzRisk then calls `OnDiffLine()` to report any added or deleted lines to each pattern. Finally, when it finishes scanning, SyzRisk calls `OnCommitEnd()` to notify the end of the commit and collect a list of matched functions from each pattern. SyzRisk also extracts the body of every changed function at this stage—both old and new versions—to prepare for function-level matching.

Matching function-level patterns. SyzRisk continues the matching procedure with function-level patterns and the extracted body of changed functions. SyzRisk first calls `GetRequiredAnalysis()` of every pattern to collect the required analysis, and after performing the analysis, SyzRisk calls `OnChangedFunc()` with every changed function to find whether the function matches patterns. After finishing all matching tasks, SyzRisk compiles the matching results as a list of all (i.e., both diff-level and function-level) matched patterns per function.

B Case Study: Weight on Root Causes

In this section, we analyze two root cause examples that were assigned with a near-median and an over-75th-percentile weight to see how typical root causes get weighted. Both examples were excerpted from $T = 30$.

B.1 Near-Median Weight

Figure 13 shows a root-cause modification example committed on Jun 2, 2020, where the pattern-based weighting method weighted the root-cause function (`mapping_set_error`) by 3.59, a near-median weight on a Jul 1, 2020 reference date. `mapping_set_error` attempted to introduce a new logic (Line 14) for a newly added struct field `s_wb_err` (Line 4), but it failed to address the case that `mapping->host` can be null. This function was indeed matched with two patterns—Chained Dereference and Modified Struct Field—that precisely describes the changes that are responsible for this root cause.

B.2 Over-75th-Percentile Weight

```
--- a/net/tls/tls_device.c
+++ b/net/tls/tls_device.c
@@ -1290,6 +1304,26 @@ int tls_device_down(struct net_dev *nd)
  list_for_each_entry_safe(ctx, tmp, &list, list) {
+  WRITE_ONCE(ctx->sk->sk_validate, tls_validate_xmit_skb_sw);
+  WRITE_ONCE(ctx->netdev, NULL);
+  set_bit(TLS_RX_DEV_DEGRADED, &ctx->flags);
+  synchronize_net();
   ...
-  WRITE_ONCE(ctx->nd, NULL);
-  synchronize_net();

   dev_put(nd);
-  list_del_init(&ctx->list);

-  if (refcount_dec_and_test(&ctx->refcount))
-   tls_device_free_ctx(ctx);
+  spin_lock_irqsave(&tls_device_lock, flags);
+  list_move_tail(&ctx->list, &tls_device_down_list);
+  spin_unlock_irqrestore(&tls_device_lock, flags);
  }
```

Figure 14: Example of an over-75th-percentile root-cause modification (memory-leak, committed on Jun 1, 2020).

Figure 14 shows a root-cause modification example committed on Jun 1, 2020, where the pattern-based weighting method weighted the root-cause function (`tls_device_down`) by 76.02, an over-75th-percentile weight on a Jul 1, 2020 reference date. The commit made extensive modifications in "`net/tls`" to mitigate a pre-existing use-after-free bug, but it created another context leak bug by illegally removing the device release logic (Line 16-17).

As extensive as the modification was, this function was matched with a total of six patterns—Chained Dereference (Line 5), Concurrent API (Line 18 and Line 20), Global Variable (`tls_validate_xmit_skb_sw` in Line 5), Locked Context (Line 19), Memory Mgmt. API (Line 16-17) and Modified Struct Field (`sk` in Line 5)—where two of them are either directly (Memory Mgmt. API) or indirectly (Modified Struct Field) related to the root cause.

C Reason of Highly Variable TTEs in Syzkaller Compared to AFL

While both Syzkaller and AFL [58] rely on randomness in nature, Syzkaller incorporates more internal randomness than AFL, resulting in a high variance of bug exposure times (i.e., TTEs). One prime example of increased randomness is the input selection algorithm; while AFL cycles through all inputs in the corpus, Syzkaller *randomly* selects an input every time. This extends the expected waiting time per individual inputs indefinitely, which results in an extremely long TTE depending on the outcome of random selection. Another example is corpus pruning (or corpus *rotation*, in Syzkaller term), where Syzkaller periodically *removes* part of the corpus in a random fashion. This also affects the high variance of TTEs, depending on which inputs are removed by corpus pruning. They are mostly to avoid wasting time by continuing with low-yielding inputs, which Syzkaller does not afford on top of the low throughput of kernel fuzzing.

D List of Keywords for Fix Commit Identification

As stated in Section 5.3.3, we identified fix commits by matching relevant keywords to commit comments. Table 5 shows the list of keywords used to identify fix commits of corresponding bug types. Every bug type was matched independently in a case-insensitive way, except `kernel-panic` that was only matched if no other bug types were matched. In case of no matches but when Syzbot reports were available, we additionally parsed the reports to extract a reported bug type in the title.

Bug Type	Keywords
use-after-free	`use.+after.+free`, `double.+free`, `uaf`
buffer-overflow	`(stack|buffer|heap|global|slab).+overflow`, `(stack|buffer|heap|global|slab|vmalloc).+overrun`, `off.+by.+one`, `out.+of.+bound`
deadlock	`dead(| |-)lock`
race-condition	`(thread|core|cpu)(1|2)` with `race`, `race\W+condition` with `(fix|prevent|mitigate)`
memory-leak	`memory.+leak`, `leak.+memory`
null-dereference	`null.+deref`, `deref.+null`
kernel-panic	`crash`, `panic`

Table 5: List of keywords for fix commit identification per bug type (in Python regular expression).

E Number of Pattern-analyzed Root Causes per Bug Type

Bug Type	Number
kernel-panic	19
buffer-overflow	26
null-dereference	30
race-condition	25
use-after-free	24
deadlock	11
memory-leak	10
uninit-value	4
kernel-infoleak	1
(Total)	150*

*: including 4 CVEs with two bug types.

Table 6: Number of pattern-analyzed root-cause commits per bug type.

The change patterns in Table 3 were derived from 38 crashing CVEs from 2020 and 2021, 65 Syzbot reports in Novemember 2020 and March 2021, and 43 additional root-cause commits in 2020 that were unmatched with initial pattern sets (Section 6). Table 6 shows the number of analyzed root-cause commits per bug type. Note that four CVEs had two bug types, resulting in the total number larger than all investigated commits by four.