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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

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...
changed every month. The excessive number of patch modifications distrusts 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 gotos, 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 fuzzer that stresses the suspected root-cause code changes. 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.

We open-source SyzRisk to facilitate pattern development and future research on kernel regression fuzzing:

https://github.com/HexHive/SyzRisk

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.

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.
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 in 2018-2022 that Syzbot could have successfully performed the root cause bisec. 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 \( f \) as the elapsed days since the modification on a given function and \( t \) 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. 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.
The modification involves several risky changes, including; To gain insight into the differentiating characteristics of root-cause developers likelier to miss some side-effect. Not only some changes are riskier than others, but 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—a group of contiguous changes—can be described as the probability of each change's risk—constituent changes simultaneously. In other words, the risk of a modification $m$ is described as;

$$ R(m) = \frac{P_{\text{bug}}(\bigwedge_{n=1}^{N} c_n)}{P_{\text{benign}}(\bigwedge_{n=1}^{N} c_n)}, $$

where $\{c_1, \ldots, c_N\}$ 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.\(^3\)

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

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 changes.

---

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 come 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).
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 during its execution.

SyzRisk decides such relevance per input based on the code weights needs to be prioritized based on the relevance to risky code changes.

SyzRisk calculating its input corpus, which already covers most kernel parts but... (meta)

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

<table>
<thead>
<tr>
<th>Callback/Parameter/Return</th>
<th>Called at:</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnCommitBegin(sha, date)</td>
<td>Beginning of commit.</td>
</tr>
<tr>
<td>OnDiffLine(v, sco, line)</td>
<td>Each changed line.</td>
</tr>
<tr>
<td>OnCommitEnd()</td>
<td>End of commit.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Callback/Parameter/Return</th>
<th>Called at:</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetReqAnalysis()</td>
<td>Before matching.</td>
</tr>
<tr>
<td>OnChangedFunc()</td>
<td>Each changed function.</td>
</tr>
<tr>
<td>GetMetadata(meta)</td>
<td>Requested metadata.</td>
</tr>
</tbody>
</table>

### 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 (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. 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 \( P_{\text{bug}}(c)/P_{\text{benign}}(c) \), where \( P_{\text{bug}}(c) \) and \( P_{\text{benign}}(c) \) 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 \( f_{\text{bug}}(c|X) \) and \( f_{\text{benign}}(c|X) \) 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 \( \hat{R}(c|X) \) if the enclosing modification satisfies the circumstantial factor \( X \) is:

\[
\hat{R}(c|X) = \frac{f_{\text{bug}}(c|X)}{f_{\text{benign}}(c|X)}.
\]  

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

- **General Risk** \( \hat{R}(c|\phi) \). This risk is the closest to the general description of a risk, where the circumstantial factor \( \phi \) is satisfied by any modification.

- **Commit-Size Risk** \( \hat{R}(c|C_{sc}) \). 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 \( sc = \lceil \log_2(s + 1) \rceil \) 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** \( \hat{R}(c|S_{sys}) \). 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. \( x^2 \) 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 \( W(F) \) is:

\[
W(F) = \text{bound}(\prod_{n=0}^{N} R(c_j)) \times 2^{-t/T}. \quad (5)
\]

Here, \( \tilde{R}(c) \) is the maximum estimated risk under all circumstantial factors considered, and \( \text{bound}(x) = -Be^{-x/10} + B \) 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.
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 \( F \) be the list of visited functions during the execution of the input \( I \). Then, the weight of the input \( w(I) \) is:

\[
w(I) = \text{avg}(W(F), VF \in F).
\]  

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\cite{64}, SyzRisk-fuzzer calculates the normalized input weight \( \omega \) against the maximum input weight in the corpus and decides the likelihood of input selection by multiplying \( 2^{(\omega - 1)} \) 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 F1xes-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-controlled Var. Mod.).
def OnDiffLine(v, sco, line):
    if (sco.type == "struct"):  
        ufld.add(line.field)
        return set().union(*[ufld[fid] for fid in mfld])
    elif (sco.type == "func"):  
        u2f[line.use].add(sco.name)
        return False
    return True

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 under-estimating 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 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.

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

Table 3: List of 23 risky change patterns.

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 (ufld 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 ufld (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 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 2b253ced712e 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

5The exception logic inside goto blocks is covered by Inside GOTO.
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 \texttt{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...
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), 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 ($\log_2(size + 1)$, 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-controlled Var. Mod. (SVM)) are generally showing high risks. Notice that the switch-related patterns are showing high risks in kernel as opposed to the relatively 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 to 2020, 2021) and considered the commits within one year from each (i.e., to Jul 1, 2018 to 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 geoman, 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.
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 can only be exposed if a bug is introduced after previous 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—more than T = 30 (0.64 vs. 0.74), suggesting that a longer fade-out constant is more effective for long-lasting bugs.

### 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.

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

---

\(^6\)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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Subsys.</th>
<th>Bug Type</th>
<th>Syzkaller</th>
<th>Average TTE (mins)</th>
<th>SyzChurn</th>
<th>SR (T=30)</th>
<th>SR (T=90)</th>
<th>Relative TTE (vs. Syzkaller)</th>
</tr>
</thead>
<tbody>
<tr>
<td>blkcg_deactivate_policy</td>
<td>block</td>
<td>deadlock</td>
<td>1,011</td>
<td>3,721</td>
<td>2,021</td>
<td>1,801</td>
<td>3.68</td>
<td>2.00</td>
</tr>
<tr>
<td>pfkey_send_acquire</td>
<td>net</td>
<td>kernel-panic</td>
<td>1,657</td>
<td>393</td>
<td>580</td>
<td>1,010</td>
<td>0.24</td>
<td>0.35</td>
</tr>
<tr>
<td>ext4_bmap</td>
<td>fs</td>
<td>deadlock</td>
<td>1,921</td>
<td>717</td>
<td>473</td>
<td>1,447</td>
<td>0.37</td>
<td>0.25</td>
</tr>
<tr>
<td>tty_port_tty_get</td>
<td>drivers</td>
<td>deadlock</td>
<td>2,307</td>
<td>1,703</td>
<td>1,802</td>
<td>1,944</td>
<td>0.74</td>
<td>0.78</td>
</tr>
<tr>
<td>jbd2_journal_lock_updates</td>
<td>fs</td>
<td>deadlock</td>
<td>2,552</td>
<td>2,655</td>
<td>716</td>
<td>3,212</td>
<td>1.04</td>
<td>0.28</td>
</tr>
<tr>
<td>nilfs_mdt_destroy</td>
<td>fs</td>
<td>use-after-free</td>
<td>3,886</td>
<td>8,501</td>
<td>6,232</td>
<td>7,578</td>
<td>1.44</td>
<td>1.06</td>
</tr>
<tr>
<td>si470x_int_in_callback</td>
<td>drivers</td>
<td>use-after-free</td>
<td>6,028</td>
<td>5,833</td>
<td>1,913</td>
<td>4,746</td>
<td>0.97</td>
<td>0.32</td>
</tr>
<tr>
<td>scl_conn_del</td>
<td>net</td>
<td>deadlock</td>
<td>7,020</td>
<td>10,080</td>
<td>6,696</td>
<td>6,220</td>
<td>1.44</td>
<td>0.95</td>
</tr>
<tr>
<td>ext4_sattr_set_handle</td>
<td>fs</td>
<td>deadlock</td>
<td>7,980</td>
<td>7,681</td>
<td>5,176</td>
<td>7,926</td>
<td>0.96</td>
<td>0.65</td>
</tr>
<tr>
<td>ntvx_attr_find</td>
<td>fs</td>
<td>buffer-overflow</td>
<td>8,178</td>
<td>9,151</td>
<td>6,380</td>
<td>4,858</td>
<td>1.12</td>
<td>0.78</td>
</tr>
<tr>
<td>vmd_insert_gfn_prot</td>
<td>sm</td>
<td>kernel-panic</td>
<td>8,544</td>
<td>6,979</td>
<td>8,221</td>
<td>6,898</td>
<td>0.82</td>
<td>0.96</td>
</tr>
<tr>
<td>detach_extent_buffer_page</td>
<td>fs</td>
<td>null-dereference</td>
<td>9,017</td>
<td>8,277</td>
<td>8,695</td>
<td>6,460</td>
<td>0.92</td>
<td>0.96</td>
</tr>
<tr>
<td>evict</td>
<td>fs</td>
<td>deadlock</td>
<td>10,080</td>
<td>3,661</td>
<td>3,675</td>
<td>2,206</td>
<td>0.36</td>
<td>0.36</td>
</tr>
</tbody>
</table>

**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).**

---

**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 Syzkaller is clear in Figure 12(a), where both of SyzRisk configurations found three to four times more unique bugs than Syzkaller. 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

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 past commits up to now.

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 OnChangefunc() 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 function 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

Figure 13: Example of a near-median root-cause modification (kernel-panic, committed on Jun 2, 2020).
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 the number of pattern-analyzed root-cause commits per bug type.

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

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel-panic</td>
<td>19</td>
</tr>
<tr>
<td>buffer-overflow</td>
<td>26</td>
</tr>
<tr>
<td>null-dereference</td>
<td>30</td>
</tr>
<tr>
<td>race-condition</td>
<td>25</td>
</tr>
<tr>
<td>use-after-free</td>
<td>24</td>
</tr>
<tr>
<td>deadlock</td>
<td>11</td>
</tr>
<tr>
<td>memory-leaf</td>
<td>10</td>
</tr>
<tr>
<td>null-dereference</td>
<td>4</td>
</tr>
<tr>
<td>kernel-infoleak</td>
<td>1</td>
</tr>
<tr>
<td>(Total)</td>
<td>150*</td>
</tr>
</tbody>
</table>

* including 4 CVEs with two bug types.

The change patterns in Table 3 were derived from 38 crashing CVEs from 2020 and 2021, 65 Syzbot reports in November 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.