Discovering Feature Flag Interdependencies in Microsoft Office

Michael Schröder
TU Wien
Vienna, Austria
michael.schroeder@tuwien.ac.at

Katja Kevic
Microsoft
Cambridge, UK
Katja.Kevic@microsoft.com

Dan Gopstein
Microsoft
New York, USA
Dan.Gopstein@microsoft.com

Brendan Murphy
Microsoft
Cambridge, UK
Brendan.Murphy@microsoft.com

Jennifer Beckmann
Microsoft
Redmond, USA
Jennifer.Beckmann@microsoft.com

ABSTRACT

Feature flags are a popular method to control functionality in released code. They enable rapid development and deployment, but can also quickly accumulate technical debt. Complex interactions between feature flags can go unnoticed, especially if interdependent flags are located far apart in the code, and these unknown dependencies could become a source of serious bugs. Testing all possible combinations of feature flags is infeasible in large systems like Microsoft Office, which has about 12 000 active flags. The goal of our research is to aid product teams in improving system reliability by providing an approach to automatically discover feature flag interdependencies. We use probabilistic reasoning to infer causal relationships from feature flag query logs. Our approach is language-agnostic, scales easily to large heterogeneous codebases, and is robust against noise such as code drift or imperfect log data. We evaluated our approach on real-world query logs from Microsoft Office and are able to achieve over 90% precision while recalling non-trivial indirect feature flag relationships across different source files. We also investigated re-occurring patterns of relationships and describe applications for targeted testing, determining deployment velocity, error mitigation, and diagnostics.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; • **Mathematics of computing → Probabilistic inference problems**.

KEYWORDS

feature flags, log analysis, causal inference, combinatorial testing

ACM Reference Format:
Michael Schröder, Katja Kevic, Dan Gopstein, Brendan Murphy, and Jennifer Beckmann. 2022. Discovering Feature Flag Interdependencies in Microsoft Office. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3540250.3558942

```
if (NEW_DESIGN && DARK_MODE) {
  reduceBrightness();
} else {
  showWhiteBackground();
  if (!RIPCORD_3456) {
    playAnimation();
  }
}
```

(a) Source code

$$\begin{tikzcd}
\text{NEW\_DESIGN} \arrow[r] \arrow[dr] & \text{DARK\_MODE} \arrow[d] \\
 & \text{RIPCORD\_3456}
\end{tikzcd}$$

(b) Interdependencies

Figure 1: Example of feature flag usage

1 INTRODUCTION

Feature flags, also known as "feature toggles," "feature switches," "feature gates," or "change gates," are a design pattern to conditionally enable a code path [12]. They are a popular method within the software industry to provide the capability to control functionality in released code. Developers can wrap new code with a feature flag which can then be dynamically toggled even after the software has been deployed. The value of a feature flag is evaluated at runtime and it is either queried from a remote location or determined based on parameters in the source code. Feature flags are used to run experiments in production (e.g., for A/B testing), to roll out features in a staged manner, or for emergency bug mitigation ("e-brakes"). In the case of an e-brake, a feature flag is toggled when faulty behaviour is observed such that the bug can be mitigated rapidly without releasing a new version of the software. For an example of how feature flags are used in source code, see figure 1a.

While feature flags enable rapid development and deployment of software systems, they can also accumulate technical debt. Managing many feature flags is complex and conflicts can result in unexpected and sometimes disastrous behaviour, as illustrated by the failure at Knight Capital Group [15], where reusing an old feature flag created erroneous trades in the stock market over a 45-minute period and resulted in the company going from one of the largest traders in US equities to becoming bankrupt. The management and complexity of feature flags increases when flags are interdependent (figure 1b). Interdependencies arise any time flags are nested, when the dynamic runtime value of one flag determines whether or not another flag is queried. In this way, code that is far downstream from the "parent" flag can be affected, and the inclusion of additional feature flags will cause yet more interdependencies. The farther apart interdependent flags are in the source code, the more indirect their relationship can be. Developers might not even be aware that some flags are interdependent, especially if the relationship extends beyond function, module, or even process boundaries. Such unknown dependencies can be (and have been) a source of serious bugs that take a significant amount of time to resolve. One way to mitigate these bugs would be to test all possible combinations of feature flags, but this quickly becomes infeasible: for the 12 000 feature flags currently active in the Microsoft Office codebase, this would amount to ~7.2×10^7 testable combinations, assuming these are all simple boolean flags—which they are not.

The goal of this research is to aid product teams to improve their system's reliability by providing a way to automatically determine feature flag interdependencies in a large software system. Knowing the relationships between feature flags that exist in a codebase provides a diversity of tangible benefits:

- We can reduce our test burden by targeting only known sets of interdependent feature flags for combinatorial testing.
- We can use the knowledge of feature flag relationships to determine the ideal deployment velocity, the speed at which changes controlled by feature flags can be rolled out.
- We can save time diagnosing failures involving feature flags by following their transitive dependencies and recognizing common interdependency patterns.
- We can prevent errors by enabling developers to check for risky dependencies before toggling a feature flag.

To this end, we developed a novel approach to analyze the feature flags that are currently active within the desktop Microsoft Office Suite. As stated, Microsoft Office currently contains around 12 000 feature flags with different life spans. Every day feature flags are being added and removed. A challenge in studying feature flag interdependencies in a large and mature system is that feature flags can occur in code written in many different programming languages. Furthermore, over the years, numerous APIs have been written to wrap the official feature flag SDK for additional requirements. The many different ways of defining feature flags in the source code, across many different programming languages, makes it hard to use static or dynamic code analysis to determine interdependencies. The novelty of our approach is that we analyze the logs that are emitted every time a feature flag is queried in a running Microsoft Office application. Assuming feature flag queries are already being logged, the passive nature of our analysis requires no changes to the surrounding configuration infrastructure and is completely decoupled from the source code itself.

We investigate the following research questions:

RQ1 How can we infer feature flag interdependencies at scale?
RQ2 What is the accuracy of our method in a real-world setting?
RQ3 Do re-occurring patterns of feature flag relationships exist?

2 RELATED WORK

Interdependent Feature Flags. The problem of interdependent feature flags is one that has existed for several decades, beginning in the world of telecommunication switching [2, 9]. The modern conundrum is well described by Rahman et al. [14], "every change to trunk should be tested across all possible combinations of enabled feature toggles. This of course introduces an explosion of tests to run." There is a common position that in practice, feature flags do not need to be exhaustively tested. Fowler [5] recommends to test only two combinations, "all the toggles on that are expected to be on in the next release" and "all toggles on," and Neely and Stolt [13] suggest that combinatorial testing can largely be ignored if the flags are independent, and these are often justified by the claim that "most feature flags will not interact with each other" [8]. In the case of Microsoft Office, however, the reality is quite the opposite. There are hundreds of interdependent feature flags, and in the course of our everyday jobs we have encountered many scenarios where undocumented and untested interactions between feature flags resulted in undesirable behavior. This unfortunate situation lead us to try to build an understanding of which feature flags were intertwined with others. This goal is difficult though, as noted by Meinicke et al. [12] who explain "finding and understanding interactions is nontrivial, especially when features are developed separately and there are no clear specifications."

Moreover the types of interaction among feature flags are complex as well. There are many ways for configuration data to be dependent on each other. Chen et al. [3] define a taxonomy of these dependencies including *Control*, *Default Value*, *Overwrite Value*, and *Behavior* dependencies. Our investigation focuses only on the *Control* dependency, where the value of one feature flag determines whether a second feature flag is or is not executed.

Mechanism of Determining Interdependency. Before studying the properties of interdependent feature flags, we first had to identify the relationships between each of the flags in Microsoft Office. Some systems, such as the one used at Facebook [16] "expresses configuration dependency as source code dependency," which entirely solves the problem of determining interdependency, however it depends on a specific infrastructure that isn't available in most systems, including ours.

For many more systems, if interdependency relationships are to be established, it must be done by inference, after the code has been written. Medeiros et al. [10] proposed a configuration-space sampling method where they used a combination of sampling algorithms to find configurations that resulted in runtime faults such as memory leaks and uninitialized variables. While they showed this technique to be valuable, it becomes either less accurate or less computationally feasible if configuration space is very large, which is the case with Microsoft Office.

A common method to analyse feature flags in the literature is to have humans validate where feature flags exist and what they're used for. This is likely a symptom of researchers needing to operate over many disparate systems that have heterogeneous feature flagging mechanisms as well as not having the same long-term incentives to automate the discovery process that the maintainer of an individual system might have. One example of manual flag discover is Meinicke et al. [11], who performed an automated search

Log	Time	Feature	Value
1	14:18:27	A	False
1	14:18:27	C	False
2	09:10:38	B	False
2	09:10:38	C	False
3	23:53:04	A	True
3	23:53:04	B	False
3	23:53:04	C	False

(a)

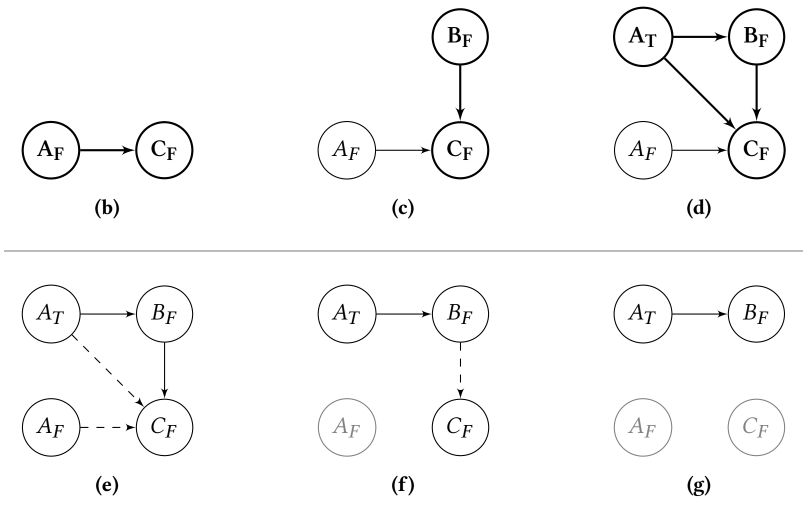


Figure 2: Using query logs (a) to discover co-occurrences (b–d) and infer causalities (e–g)

through Git commit messages to find repositories which likely contained feature flags, but then used manual inspection to verify the flags existed. A system the size of Microsoft Office is too large for this approach, and instead the relationship between configuration values must be discovered as an automated process.

The bulk of research on feature flag or configuration interdependency is done in a static analysis context. For example, Zhang et al. [17] use static analysis to analyze which regions of code are effected by configuration options, and from that determine which configurations depend on each other. Their goal was specifically to find "silent misconfigurations," configuration values which have no effect on the running program, often due to interactions between configuration settings. Static analysis has many benefits including well-defined correctness guarantees and the ability to find potential future problems before they're executed. Conversely, it is difficult to have a static analysis system that can seamlessly process unconventional systems such as dynamically generated/loaded code, programs that use multiple languages, and even large projects in a single language that are only able to be built using complex compiler configuration that is difficult to replicate in an external system.

Despite not being popular for investigating interdependency, runtime analysis has proven useful in many contexts related to independent configurations. For example, Attariyan and Flinn [1] use dynamic information flow analysis to trace data coming from configuration files to eventual errors as a tool for automated configuration debugging. Given the complexity of the Microsoft Office engineering ecosystem, we opted for the more robust option of dynamic analysis on which to base our investigation.

3 INFERRING RELATIONSHIPS

For any two feature flags A and B, we want to determine whether the value of A determines if B is queried. In particular, we want to determine if "A causes B," i.e., $A \longrightarrow B$, or if the value of A has no effect on whether B is queried, i.e., $A \nrightarrow B$.

For example, the `DARK_MODE` flag in figure 1 is only queried if the value of the `NEW_DESIGN` flag is true (assuming short-circuiting of logical operators), so `NEW_DESIGN` $\longrightarrow$ `DARK_MODE`. However, whether or not `DARK_MODE` is queried is independent of the value of the `RIPCORD_3456` flag, so `RIPCORD_3456` $\nrightarrow$ `DARK_MODE`.

Sometimes, feature flag relationships are easily inferable from the source code itself. In general, however, the heterogeneous nature of a large codebase makes static analysis difficult, especially for non-local relationships. Feature flags might be spread across different compilation units or be only very indirectly related. In these cases, we have to resort to dynamic analysis of the code's actual runtime behaviour. Fortunately, it is possible to do this in an entirely passive manner, without changes to the source code. In Microsoft Office, any time a feature flag is queried during the run of an application, the query is logged, together with the current value of the flag. Figure 2a presents a simplified example of such query logs. By combining the logs from multiple runs exercising different parts of an application, we can gain broad insight into global feature flag activation patterns.

3.1 Co-Occurrence Discovery

If $A \longrightarrow B$, then we would expect the timespan $\Delta_{AB} = t_B - t_A$ between any particular query of A (at time t_A) and the following query of B (at time t_B) to always be roughly the same, for all instances of A and B that occur in the logs. The actual value of Δ_{AB} will be different for every pair of related feature flags and could range anywhere from a few nanoseconds (e.g., for flags that occur on the same line of code) to even a few seconds (e.g., for flags that are related via some asynchronous operation, like copy-paste).

We can view Δ_{AB} as a relative measure of similarity between the contexts in which flags A and B are evaluated. For example, two flags that are queried in a single expression on the same line of source code have very similar evaluation contexts, and thus a small Δ_{AB}, as will two flags that are located in entirely different source files but connected via a function call; however, two flags that are queried at entirely different points during an application's run will have a large Δ_{AB}, regardless of whether they are spread far apart in the source code or appear within a few lines of each other.

We can collect all *co-occurring* feature flags by dragging a sliding window of some empirically determined size Δ over the query logs, selecting all feature flag pairs with $\Delta_{AB} \leq \Delta$. Figures 2b to 2d demonstrate this process (with $\Delta = 1\,\mathrm{s}$) and show how a graph representation of the discovered co-occurrences is successively built up. In this co-occurrence graph, each vertex represents a feature flag query that returned a particular value (A_F meaning flag A with value *False*) and each edge signifies that the two connected queries co-occurred within the same time window Δ. Note that the edges are directed: we take the temporal order of queries into account to avoid adding obviously paradoxical relationships—if A is queried before B, then $B \nrightarrow A$.

Algorithm 1 shows the co-occurrence discovery process in detail. Although the resulting co-occurrence graph already significantly reduces the state space of possible relationships (cf. section 4.1), it of course includes many co-occurrences that are merely coincidental and not actual causal relationships. To discover those, we need to employ causal reasoning.

Algorithm 1: Co-Occurrence Discovery

```
Input: set of feature query log files $L$; time window size $\Delta$
Output: co-occurrence graph $G = (V, E)$
let $G = (V, E)$ be an empty directed graph;
for each log file $L$ do
    for each sliding time window $W$ of size $\Delta$ in $L$ do
        for each feature query $q$ in $W$ do
            if $q \in V$ then
                increase the count of $q$ in $V$ by 1;
            else
                add $q$ to $V$ with an initial count of 1;
        for each 2-combination $(q_1, q_2)$ in $W$ do
            if $(q_1, q_2) \in E$ then
                increase the count of $(q_1, q_2)$ in $E$ by 1;
            else
                add $(q_1, q_2)$ to $E$ with an initial count of 1;
```

3.2 Naive Causal Reasoning

To turn a co-occurrence graph into a *causal graph*, whose vertices represent single feature flags and whose directed edges indicate causal parent-child relationships, we must look at the *values* of prospective parent flags. The main intuition is that if B is queried regardless of the value of A, then $A \nrightarrow B$.

To illustrate this, figures 2e to 2g proceed with the running example and successively eliminate non-causal edges from the co-occurrence graph. First, the edges $A_T \longrightarrow C_F$ and $A_F \longrightarrow C_F$ are removed (figure 2e), because if both A_T and A_F co-occur with C_F, then neither can actually be a causal factor for C; the value of A is clearly immaterial to whether or not C is queried.

Next, $B_F \longrightarrow C_F$ is removed (figure 2f), because even though we do not see $B_T \longrightarrow C_F$, we also do not have any knowledge of $B_T \nrightarrow C_F$, as B_T does not occur at all. Merely knowing of a co-occurrence ($B_F \longrightarrow C_F$) is not enough evidence for us to assume a causal relationship ($B \longrightarrow C$), we also require evidence of the absence of counter-evidence ($B_T \nrightarrow C_F$). Put another way: in order to determine that some feature flag is the parent of another, we need to see both the cases where the flag is (or could be) the parent, and the cases where it is not. It is only by contrasting these two scenarios that we can gain information.

Finally, only $A_T \longrightarrow B_F$ remains (figure 2g) and thus the causal graph is simply $A \longrightarrow B$.

3.3 Noise

Figure 3 shows a typical instance of a real-world co-occurrence graph. Naive causal reasoning would require us to eliminate all of its edges, because they clearly contradict one another. But not all of the (co-)occurrences in this graph are equally valid; some of them are purely *noise*, which can appear for a number of reasons:

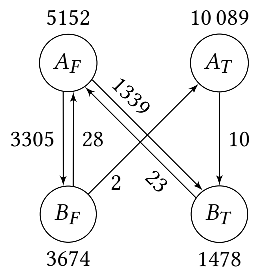
Bugs Logging feature flag queries happens in a variety of heterogeneous environments and involves local caching, asynchronous batched network transmissions, and server-side log processing. Bugs can and do happen: queries get dropped or logged in duplicate, time-ordering gets mixed up, and so on. While we could work under the assumption that bugs are relatively rare and could be mitigated by rigorously cleaning our input data, we would much prefer to be able to draw valid conclusions from data that occasionally includes small, inexplicable amounts of noise. Such is the nature of industrial software engineering.

Crossed Signals Our logs contain feature flag queries across a variety of apps on a variety of platforms. Some of these share the same feature flags but use them in different ways, exhibiting different interdependencies. It certainly makes sense to process some subsets of our logs separately, e.g., partitioned by platform. On the other hand, since apps do communicate with each other and there are legitimate feature relationships that cross app boundaries, we would also like to capture those.

Code Drift As the source code changes over time, and feature flags are added and removed, the relationships between feature flags change as well. The query logs are like a slow moving window sliding over the released app versions, capturing multiple versions at once and slightly lagging behind the latest changes in the source code, but steadily catching up. As most relationships between feature flags remain relatively stable, however, limiting ourselves to only logs from the very latest (released or unreleased) app versions would severely limit the amount of data available for analysis.

Coincidences Sometimes the data just lines up in a way that is indistinguishable from a real signal. In principle, we will never be able to entirely rule out this kind of noise. In practice, we would like our analysis method to be sensitive enough to discard many, if not most, such coincidences.

While some sources of noise can be mitigated, we would like to deal with most data as-is. How can we infer causal relationships in the presence of noise? And how can we be confident that our inferences are correct, given that one small change in signal could completely change the result?



$\boxed{A \xrightarrow{?} B}$

$$P(B \mid A_T) = 10/10089 \approx 0.00$$
$$P(B \mid A_F) = (3305 + 1339)/5152 \approx 0.90$$
$$P(A_T \mid B) = 10/(3674 + 1478) \approx 0.00$$
$$P(A_F \mid B) = (3305 + 1339)/(3674 + 1478) \approx 0.90$$
$$E_T = ((1 - 0.00) + 0.90 + (1 - 0.00) + 0.90)/4 = 0.95$$
$$E_F = (0.00 + (1 - 0.90) + 0.00 + (1 - 0.90))/4 = \mathbf{0.05}$$

$\boxed{B \xrightarrow{?} A}$

$$P(A \mid B_T) \approx 0.02$$
$$P(A \mid B_F) \approx 0.01$$
$$P(B_T \mid A) \approx 0.00$$
$$P(B_F \mid A) \approx 0.00$$
$$E_T \approx 0.50$$
$$E_F \approx 0.50$$

Figure 3: Causal inference on a noisy real-world co-occurrence graph. The scenario $A_F \longrightarrow B$ is the most likely one, because its error value $E_F = 0.05$ is the smallest, indicating the least deviation of its probabilities from the expected pattern.

3.4 Causal Reasoning with Probabilities

To deal with noisy data we can borrow some notions from probability theory. We can view a feature flag query as a *random event* that either occurs within some time window or does not. We can also view the value of the feature flag as the *outcome* of the event. The probability that some feature flag B co-occurs with another feature flag A when the value of A is x can then be described by the *conditional probability*

$$P(B \mid A_x) = \frac{P(A_x \cap B)}{P(A_x)} = \frac{\text{co-occurrences of } A_x \text{ with } B}{\text{total occurrences of } A_x}.$$

Assuming that we know A_x occurs before B, this can be interpreted as: "How likely is it that B will be queried if A has the value x?" The inverse—"How likely is it that A had the value x if we know that B was queried?"—is given by

$$P(A_x \mid B) = \frac{P(A_x \cap B)}{P(B)} = \frac{\text{co-occurrences of } A_x \text{ with } B}{\text{total occurrences of } B}.$$

If A has k possible (observed) values, then there are $2k$ such probabilities between A and B, assuming we know that A comes before B (and thus the value of B is not relevant). But how do these probabilities help us determine whether $A \longrightarrow B$ or $A \nrightarrow B$?

Let us consider the platonic ideal of a causal relationship:

```
if (A) {B}
```

Here, A is assumed to be a boolean flag and this is the only occurrence of both A and B in the source code. Clearly, the likelihood that B will be queried if A is true is 100 %, while the likelihood that B will be queried if A is false is 0 %. Similarly, the likelihood that A was true if B is queried is 100 % and the likelihood that A was false if B is queried is 0 %. We observe $P(B \mid A_T) = 1$, $P(B \mid A_F) = 0$, $P(A_T \mid B) = 1$, and $P(A_F \mid B) = 0$.

Realistically, A or B might occur multiple times in the source code, possibly in relation with other feature flags:

```
if (A) {X}        if (A && X) {B}
if (X) {B}        if (X || A) {B}
```

The probabilities between A and B will then be affected by some values proportional to the number of additional children of A and additional parents of B. In particular, we now have $P(B \mid A_T) = 1-\alpha$, where α is some term proportional to the number of additional children of A, and $P(A_T \mid B) = 1 - \beta$, where β is some term proportional to the number of additional parents of B.

The table below gives the expected probabilities for the three possible scenarios: $A_T \longrightarrow B$, which we just discussed; the complementary $A_F \longrightarrow B$, i.e., replacing `A` by `!A` in the code; and the case when neither A_T nor A_F are a cause of B and thus $A \nrightarrow B$.

	$A_T \longrightarrow B$	$A_F \longrightarrow B$	$A \nrightarrow B$
$P(B \mid A_T)$	$1 - \alpha$	0	ε_1
$P(B \mid A_F)$	0	$1 - \alpha$	ε_2
$P(A_T \mid B)$	$1 - \beta$	0	ε_3
$P(A_F \mid B)$	0	$1 - \beta$	ε_4

In the case of $A \nrightarrow B$, the probabilities are unknown random values ε_1 to ε_4, about which we know nothing, except that they are very unlikely to match the probabilities we expect in the other two cases. The exact values of α and β are also unknown, and they are different for each particular combination of feature flags A, B, and X, but it is reasonable to assume that for most feature flags the number of parents and children will be much closer to one than, for example, ten. Both α and β are thus expected to be significantly smaller than one on average.

Knowing which probabilities to expect for $A_T \longrightarrow B$ and $A_F \longrightarrow B$, we can calculate two *error values* E_T and E_F, indicating how much reality deviates from the expectations for each scenario. The smaller the error, the more likely the scenario; if both errors are too large, then $A \nrightarrow B$. Figure 3 demonstrates these calculations on a noisy graph based on real data. In the remainder of this section, we formalize this idea and generalize it to non-boolean flags.

Probabilistic Causal Inference. Assume that A and B are feature flags, with A having k observed values, and that A occurs before B. As a shorthand, we will write A_i for the total number of occurrences of A that return value i, B for the total number of occurrences of B (returning any value), and A_iB for the number of co-occurrences of A_i and B. For each of the k possible scenarios $A_i \longrightarrow B$, we can compute an error value

$$E_i = \frac{1}{k+2}\left(\left(1 - \frac{A_iB}{A_i}\right) + \sum_{j \neq i}^{k} \frac{A_jB}{A_j} + \left(1 - \frac{A_iB}{B}\right) + \sum_{j \neq i}^{k} \frac{A_jB}{B}\right).$$

The overall error for the possibility $A \longrightarrow B$ is then given by

$$E = \min_i^k E_i.$$

Because E only captures the *relative* proportions between A and B, we assess our confidence in E by computing the least *absolute* number of contributing observations

$$N = \min(A_1, \ldots, A_k, B)\,.$$

Then, for empirically determined thresholds $\hat{E}$ and $\hat{N}$,

$$\begin{aligned} A &\rightarrow B \quad \text{if } k \geq 2 \text{ and } E \leq \hat{E} \text{ and } N \geq \hat{N}, \\ A &\nrightarrow B \quad \text{otherwise.} \end{aligned}$$

We are thus able to infer interdependence between feature flags based on observed (co-)occurrences.

RQ1. How can we infer feature flag interdependencies at scale? Looking solely at *query logs*, we are able to discover feature flags that repeatedly *co-occur* within certain time windows. Based on intuitions about code structure and employing notions from probability theory, we developed a method of *probabilistic causal reasoning* that is robust to noise by calculating how closely a pair of co-occurring feature flags matches an ideal causal relationship.

4 EVALUATION

We implemented our inference mechanism in Python and applied it to real-world feature flag query logs from Microsoft Office. We chose a sub-sample of query logs restricted to a single release platform and code fork, which made it easier to cross-reference potential findings with the codebase. For a period of one week, we collected about 2.5 million feature queries per day, from about 80 000 daily app sessions. We performed co-occurrence discovery every day, with a time window size $\Delta = 1\,\mu s$, incrementally updating our database of co-occurrences and re-calculating all causal probabilities afterwards. At the end of the collection period, we had discovered 5 946 317 pairs of 12 791 co-occurring feature flags. Of these, 326 418 pairs of 5724 feature flags are potentially causally related ($E \leq 0.50$) and 593 pairs of 612 feature flags were considered to be likely causally related ($E \leq 0.25$). Figure 5 presents some concrete examples of found relationships.

4.1 Precision

To evaluate the *precision* of our approach—how many of the relationships we uncover are actually true causal relationships?—we cross-checked the results of our inference algorithm with the Microsoft Office source code. We selected 200 pairs of 327 feature flags in a purposive sample covering the range of E and N values returned by our algorithm. The sample is balanced, with 107 of the sample pairs exhibiting a real causal relationship in the codebase, and 93 of no discernible causality. We manually inspected the source code locations of each selected feature flag pair to determine causality. This was a time-consuming process, as it is often not immediately apparent whether a causal relationship exists, especially for relationships that would be rather indirect. We erred on the side of caution, and only reported true positives when the causal relationship was clear beyond doubt; if the examiner was not able to establish a causal relationship after some time (typically about 15 minutes), the feature flag pair in question was marked as a false

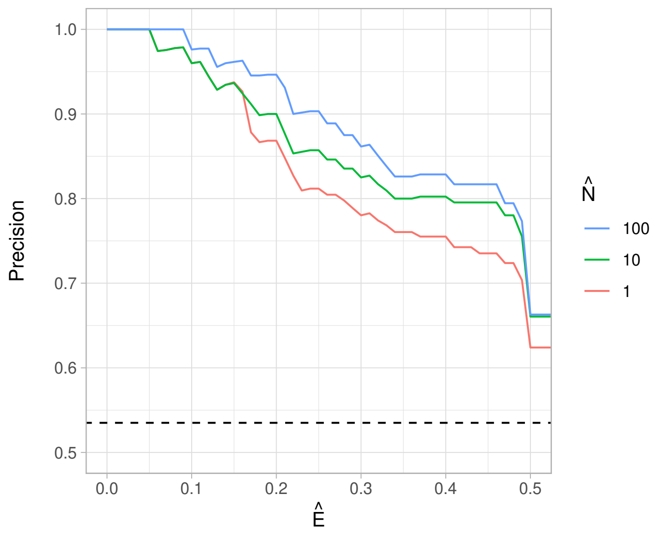


Figure 4: Precision for different values of $\hat{E}$ and $\hat{N}$. As our willingness to accept unlikely candidates increases, so do the rates of false positive parent-child relationships.

Table 1: Precision for $\hat{N} = 100$ and different values of $\hat{E}$

	Discovered		Verified		Falsified		
$\hat{E}$	Pairs	Flags	Pairs	Flags	Pairs	Flags	Precision
0.01	16	31	7	14	0	0	1.00
0.05	98	167	28	50	0	0	1.00
0.10	149	231	41	72	1	2	0.98
0.15	214	296	50	87	2	4	0.96
0.20	305	372	53	92	3	6	0.95
0.25	593	612	56	96	6	12	0.90
0.30	941	901	56	96	9	18	0.86
0.35	2358	1791	57	98	12	24	0.83
0.40	7130	3012	58	99	12	24	0.83
0.45	10 247	3430	58	99	13	26	0.82
0.50	326 418	5724	59	99	30	57	0.66

positive—thus it is possible that the number of true positives is actually higher than what we report.

Figure 4 shows the precision (true positives divided by sample) plotted against $\hat{E}$, for different choices of $\hat{N}$. We are able to achieve 100% precision with $\hat{E} = 0.05$ (regardless of $\hat{N}$), and 90% precision with $\hat{E} = 0.25$ and $\hat{N} = 100$. The exact numbers of manually verified (true positive) and falsified (false positive) pairs are given in table 1, which also shows how many pairs of feature flags we are able to discover at different levels of $\hat{E}$.

Choosing $\hat{E} = 0.50$ and $\hat{N} = 100$, i.e., classifying rather unlikely pairs to be related, we still achieve a precision of 66%—significantly better than chance. This makes sense, because the co-occurrence discovery step already reduces the set of possible relationships in a major way, filtering out those pairs of feature flags which are definitely not related. Adding probabilistic causal reasoning on top, i.e., only counting pairs with a $E \leq 0.50$, naturally increases precision further.

AugLoopRuntime.cpp

```
bool FSimilarityEnabled() {
  static const FeatureFlag EduEnabled {...};
  static const FeatureFlag ConEnabled {...};
  static const FeatureFlat EntEnabled {...};
  return (EduEnabled || ConEnabled || EntEnabled);
}
```

$$\begin{tikzcd} \text{EduEnabled} \arrow[dr] \arrow[rr] & & \text{ConEnabled} \arrow[dl] \\ & \text{EntEnabled} & \end{tikzcd}$$

(a) Triangular relationship

EntityManager.cpp

```
void EntityManager::Init() {
  if (FeatureFlags::Instance(m_pWorkbook).AutoRefresh()) {
    RefreshManager::CreateSharedInstance(m_pWorkbook);
  }
}
```

RefreshManagerImpl.cpp

```
void RefreshManagerImpl::CreateSharedInstance(Workbook* pWorkbook) {
  try {
    refreshManager = GetApi<RefreshManager>(NEWSHAREDOBJ(
        RefreshManagerImpl, pWorkbook));
  } CATCH_HANDLER
}

RefreshManagerImpl::RefreshManagerImpl(Workbook* pWorkBook) :
  m_pWorkbook(pWorkbook),
  m_fRefreshBar(FeatureFlags::Instance(pWorkbook).ShowRefreshBar()),
  ...
```

$$\begin{tikzcd} \text{AutoRefresh} \arrow[d] \\ \text{ShowRefreshBar} \end{tikzcd}$$

(b) Indirect relationship across multiple files

Word.xml

```
<FSDropGallery Id="flyoutInsertPics" FeatureFlag="PictureRibbon">
 <Commands>
  <FSMenuCategory Class="StandardItems">
   <Items>
    <FSExecuteAction Id="insertPicFromFile" />
    <FSExecuteAction Id="insertOnlinePic" FeatureFlag="OnlinePics" />
    <FSExecuteAction Id="clipArtDialog" />
   </Items>
  </FSMenuCategory>
 </Commands>
</FSDropGallery>
```

$$\begin{tikzcd} \text{PictureRibbon} \arrow[d] \\ \text{OnlinePics} \end{tikzcd}$$

(c) Relationship in resource file

Figure 5: Real causal relationships between feature flags found in the Microsoft Office codebase. The source code has been simplified for presentational purposes.

4.2 Recall

Since our goal is to find relationships between feature flags that are as-of-yet unknown, we do not have *a priori* ground truth. This makes it difficult to establish *recall*—how many of the relationships between feature flags that are hidden in the codebase can our method uncover? We are unable to answer this question directly. However, we can make inferences based on the quality of our results; in particular, the types of relationships we are seeing.

Figure 5a is an example of an "obvious" relationship: three feature flags are queried together as part of a boolean predicate, giving rise to a triangular interdependency; only if the EduEnabled flag is false will the ConEnabled flag be queried, and only if both EduEnabled and ConEnabled are false will EntEnabled be queried. This relationship is manifested entirely in a single line of source code, producing a strong signal in the query logs that our system can easily detect.

Figure 5b shows a much more indirect relationship, spanning multiple source files. Here, the parent (AutoRefresh) and child (ShowRefreshBar) flags are queried in different program modules and are separated in the control flow by a number of function calls involving macro expansions, class constructors, and C++ templates. A purely static approach might have some difficulties with this, but our log-based analysis naturally captures the dynamic control flow; the surrounding syntactic complexity is entirely irrelevant.

Figure 5c demonstrates that our approach is also completely language-agnostic. In addition to flag usage in C++, C#, and other programming languages, we are able to find dependencies between feature flags used solely in non-code resource files, as in the present case of the PictureRibbon → OnlinePics pair found in an XML configuration file used to construct an application UI.

Given the diversity of relationship types we are able to find (see also section 5), including very indirect relationships, we believe that our results are indicative of non-trivial recall.

RQ2. What is the accuracy of our method in a real-world setting? To determine the *precision* of our approach, we manually evaluated a subset of discovered relationships in a large-scale real-world codebase and found that we are able to achieve 90% precision for likely pairs ($E \leq 0.25$), with an absolute minimum precision of 66%. While we are unable to precisely quantify *recall* due to a lack of ground truth, we see evidence of non-trivial recall in the indirect nature of some of the discovered relationships, which can span multiple files and programming languages.

5 INTERDEPENDENCY PATTERNS

So far, we have discussed feature flag relationships mostly as pairwise parent-child relationships between two flags. As figure 5a demonstrates, more complex patterns can emerge once transitive dependencies are taken into account. Each of the two flags in a parent-child relationship can themselves be in further parent-child relationships with other flags (which is reflected in the values of α and β in section 3.4). To investigate the extent of such transitive interdependencies and whether or not they give rise to re-occurring patterns, we can study the *global causal graph* of feature flags, as seen in figure 6. Here, we plotted the 612 feature flags from our evaluation (section 4) that were considered to be likely causally related ($E \leq 0.25$). Nodes correspond to feature flags and the directed edges represent parent-child relationships. The weakly connected components of this graph are *feature clusters*, i.e., subsets of feature flags that are only (indirectly) connected to each other but not to flags from any other subset. The layout was achieved using the Fruchterman-Reingold algorithm [6], which naturally brings out independent clusters. The location and distance of nodes hold no further meaning.

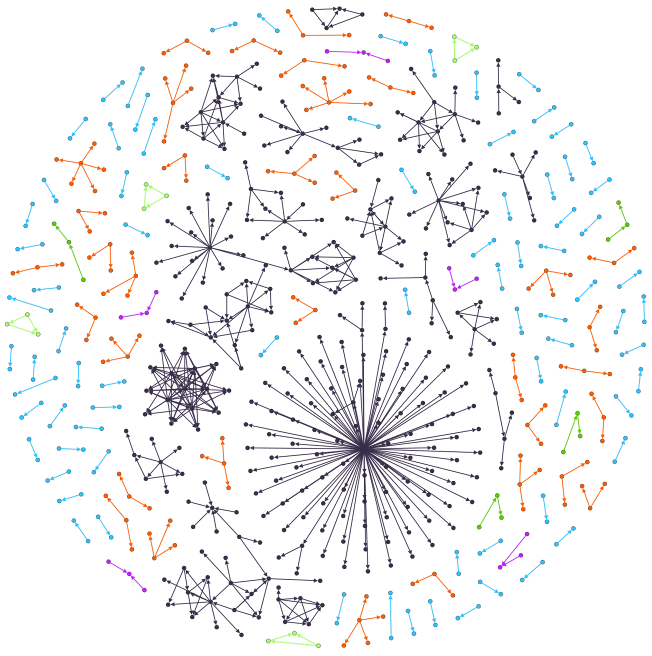


Figure 6: Causal graph of all feature flags (90% precision), showing 146 feature clusters: 79 simple pairs, 35 outward stars, 5 inward stars, 4 chains, 4 triangles, 19 other kinds.

Based on visual inspection of this graph, we identified five basic patterns of feature flag interdependencies. The identified patterns, the rules used to determine if a feature flag cluster belongs to a specific pattern, as well as examples of code structures that could give rise to each pattern, are given in table 2.

The most common pattern is the *simple pair* of parent-child flags, occurring 79 times in our sample and involving 158 flags (25.8% of all flags in the sample). The second most common is the *outward star* pattern, involving 122 flags (19.9%), where one parent flag is at the center of numerous parent-child relationships, but the children are themselves not interconnected. This situation arises when a single flag guards a large section of code containing many independent flags, or when a (often non-boolean) feature flag acts as a configuration parameter that is repeatedly used in scenarios involving other flags. Less common, involving only 15 flags (2.5%), is the *inward star*, where a child flag has multiple parent flags, which can occur when the child flag is reused in different code contexts. *Triangle* and *chain* patterns each only occur 4 times in our sample,

Table 2: Identified patterns of feature flag interdependencies

Pattern	Description	Code Example	Occurrence	Involved Flags

Chain
$\bullet \to \bullet \to \cdots \to \bullet$
	At least three nodes that are in consecutive parent-child relationships.	`if (A) {B}`
		`...`
		`if (B) {C}`
			4 (2.7%)	12 (2%)

Triangle $\begin{tikzcd} \bullet \arrow[r] \arrow[rr, bend left] & \bullet \arrow[r] & \bullet \end{tikzcd}$	At least three nodes in a chain, with the first node also being the parent of the last node.	`(A && B && C)`	4 (2.7%)	12 (2%)

Inward Star
$\bullet \to \bullet \leftarrow \bullet$
	One node is the child of at least two parents, which are not themselves connected.	`if (A) {C}`
		`if (B) {C}`
			5 (3.4%)	15 (2.5%)

Outward Star
$\bullet \leftarrow \bullet \to \bullet \to \bullet$
	One node is the parent of at least two children, which are not themselves connected.	`f(A,B);`
		`g(A,C);`
			35 (24%)	122 (19.9%)

Simple Pair $\bullet \to \bullet$	Two nodes that are in a parent-child relationship.	`if (A) {B}`	79 (54.1%)	158 (25.8%)
Other	Unclassifiable; often basic patterns with slight deviations, or superclusters of multiple patterns.		19 (13%)	293 (47.9%)

and are closely related: triangle formations are usually due to short-circuiting boolean predicates or closely nested if statements, while chains arise either when consecutive parent-child relationships are not nested but purely sequential, or when the relationships are very indirect, with enough distance between parent and grandchild to not be recognized as a triangle.

In addition to these basic patterns, a number of clusters remained unclassifiable (19 out of 146, involving 293 flags in total). Of these, many are essentially one of the basic patterns with slight deviations preventing easy classification. For example, one large cluster involving 102 flags (the "starburst" in the lower center of figure 6) is almost a pure outward star pattern, save for a few interconnected children. Other unclassifiable patterns arise when two or more basic patterns are connected by a *bridge node*, forming a singular supercluster. Bridge nodes could indicate two otherwise unrelated application components that are linked by a common feature flag, increasing software coupling and perhaps introducing a hidden interdependency. Inability to assign one of the basic classifications may well be an indicator of unusual complexity and therefore risk.

RQ3. Do re-occurring patterns of feature flag relationships exist? We found five re-occurring patterns of feature flag interdependency relationships: simple pairs, outward stars, inward stars, triangles, and chains. Other types of feature flag clusters are often deviations from these basic patterns. We can use interdependency patterns to identify unusual or risky code structures.

6 THREATS TO VALIDITY

While our work is based on real world data of a large-scale and mature software system, there are threats to the generalizability of of our approach.

Idealized Assumptions. If the relationships between feature flags are actually significantly different than the platonic ideal `if(A){B}`, or the average number of children and parents per feature flag (reflected in the values of α and β) much higher than we assume, then our probabilistic method might have a hard time inferring relationships. However, we based our assumptions on our direct experience with actual code containing feature flags and empirical evaluation confirms the effectiveness of our approach.

Lack of Ground Truth. We have mentioned the difficulty of establishing *recall*, as we lack ground truth. It is possible that our approach, while able to find some relationships, is still missing a significant number. But based on our findings, which *do* include non-trivial indirect relationships, we are confident of achieving reasonable recall. The parameters ($\Delta, \hat{E}, \hat{N}$), which influence recall, need to be chosen empirically and we believe we made reasonable choices for the purposes of this paper; we have limited evidence that by increasing Δ we can further improve recall (see section 7).

Cold Start Problem. Our approach is fundamentally data-driven: in order to make inferences about possible relationships between feature flags, the data needs to contain evidence of these relationships, in the form of sequential feature queries; to generate these feature queries, the applications need to run with certain combinations of feature flags enabled; without knowing the relationships between feature flags beforehand, we would need to test all possible combinations of flags, with all possible values, in order to generate the data necessary to make complete inferences—this is computationally infeasible. In reality, for our applications, we do not actually need to have perfect recall. Being able to infer a significant amount of interesting relationships is enough to make the system useful. Furthermore, preliminary inference results can be used to selectively generate missing data, enabling more inferences and improving recall (see section 7).

Codebase Bias. If the inference mechanism is too closely tailored to the particularities of a single codebase (i.e., that of Microsoft Office) and the uses of feature flags therein, then it might not be transferable to other applications. However, we believe that the foundations of our approach are entirely application-agnostic and that it is sufficiently general to be applicable to other codebases. Moreover, Microsoft Office itself consists of a heterogeneous set of applications, with massive differences between their individual core components.

7 FUTURE WORK

In the future, we aim to improve both precision and recall by completing our dataset and investigating larger time windows; and we want to further explore patterns of interdependencies among feature flag clusters.

Completing the Dataset. The probabilistic causal discovery approach works best with complete data, i.e., a dataset in which both boolean feature flag values are present. As the dataset in practice is oftentimes incomplete, i.e., only one feature flag value is present as opposed to both, we plan to systematically run an automated test suite [7] on Microsoft Office applications with different sets of feature flag values. The output that is logged by the simulator in our test suite is exactly the same as when real users would use a Microsoft Office application.

Investigating Larger Time Windows. We plan to evaluate our approach using larger co-occurrence time windows (Δ), which could allow us to capture feature flag pairs that are being queried further apart. We hypothesize that more nested feature flags might be discovered in features that take longer to fully execute due to user interactions, e.g., copying and pasting.

Exploring More Interdependency Patterns. Feature flag pattern recognition could be improved by tolerating slight deviations from existing patterns and by recognizing more complex combinations, identifying bridge nodes and superclusters. We also want to better understand what code structures give rise to which interdependency patterns, and how such patterns are linked to faults.

8 APPLICATIONS

We performed this research in response to several practical problems we regularly face in our organization. One of the most valuable outcomes of this work is the diversity of tangible benefits we can receive by applying our findings. These issues span the entire lifecycle of our product, from automated testing to client-side error mitigation. Further, the challenges we hope to address have impacts that range from increased organizational efficiency to simplified development practices.

Targeted Testing. Testing all possible combinations of feature flag values becomes substantially harder as more feature flags are used. The combinatorial explosion that occurs when using many feature flags makes it impossible to test all combinations. Fowler [5] recommends to test the feature flags that are known to be enabled in the next release. However, large projects can contain thousands of feature flags where every flag can be toggled. Therefore, it is important to enable tooling that helps to systematically test only the relevant combinations. Our research on feature flag co-occurrences can be applied to substantially decrease the number of feature flag value combinations to test, as only the co-occurring feature flags' combinations need to be targeted for combinatorial testing. Flags that are not co-occurring can be tested independently of each other. Conversely, flags which are discovered to be involved in complex relationships can be highlighted for additional scrutiny.

Deployment Velocity. We plan to use the knowledge of feature flag dependencies to determine the velocity with which a flag can be rolled out. Feature flags, by their design, indicate the usage of unique modules of code. Interdependent features then indicate interdependent modules, which is the main factor in coupled code. It is well studied that software coupling is correlated with negative quality indicators, such as vulnerabilities [4]. Consequently, we extrapolate that interdependent flags are more at risk of admitting vulnerabilities. We can use this information to roll out changes slower to ensure that they're thoroughly understood and tested before being fully deployed.

Diagnostics. Failures rooted in feature flags can be tedious and time-consuming to diagnose. Troubleshooting failures when multiple feature flags are involved can incur substantial costs [1]. Showing explicitly which feature flags are interdependent has the potential to decrease the time to mitigate the problem, and it might uncover previously unknown relationships as the cause of failure.

Error Mitigation. Many features are developed behind feature flags, such that the flag can be toggled in case of a failure [14]. The typical response to discovering an error behind a feature flag is to mitigate the error by immediately disabling the flag. In the case of interdependent flags, however, this can have unintended side effects. It could disable more features than intended, or leave the system configuration in an unexpected and untested state. Our research can enable developers to check if there are any dependencies before toggling a feature flag, which can help to prevent a further regression.

9 CONCLUSION

In this paper, we described an approach for automatically discovering interdependencies between feature flags in order to aid product teams in improving their system's reliability. Unknown dependencies between feature flags can be a source of serious bugs but testing all possible flag combinations is infeasible for large projects. Our approach is based solely on analyzing feature flag query logs and is especially suited for large heterogeneous codebases. We developed a method of probabilistic causal reasoning that is language-agnostic and robust against noise. We applied our approach on the Microsoft Office codebase and achieved high precision and non-trivial recall. In analysing the results, we found patterns of feature flag relationships that can be indicators for the amount of risk associated with certain flags. Our work can be applied in reducing the test burden for combinatorial testing, in determining deployment velocity for safe rollouts, in diagnostics of faults involving feature flags, and in error mitigation by preventing regressions. In the future, we will use automated testing to increase and improve the data available for analysis and we plan to experiment with different time windows to discover a wider range of possible relationships.

REFERENCES

[1] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *OSDI*, Vol. 10. 1–14.

[2] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks* 41, 1 (2003), 115–141. https://doi.org/10.1016/S1389-1286(02)00352-3

[3] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 362–374. https://doi.org/10.1145/3368089.3409727

[4] Istehad Chowdhury and Mohammad Zulkernine. 2010. Can Complexity, Coupling, and Cohesion Metrics Be Used as Early Indicators of Vulnerabilities?. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (Sierre, Switzerland) *(SAC '10)*. Association for Computing Machinery, New York, NY, USA, 1963–1969. https://doi.org/10.1145/1774088.1774504

[5] Martin Fowler. 2010. *FeatureToggle*. https://martinfowler.com/bliki/FeatureToggle.html

[6] Thomas M. J. Fruchterman and Edward M. Reingold. 1991. Graph Drawing by Force-Directed Placement. *Softw. Pract. Exper.* 21, 11 (nov 1991), 1129–1164. https://doi.org/10.1002/spe.4380211102

[7] Luke Harries, Rebekah Storan Clarke, Timothy Chapman, Swamy V. P. L. N. Nallamalli, Levent Özgür, Shuktika Jain, Alex Leung, Steve Lim, Aaron Dietrich, José Miguel Hernández-Lobato, Tom Ellis, Cheng Zhang, and Kamil Ciosek. 2020. DRIFT: Deep Reinforcement Learning for Functional Software Testing. *CoRR* abs/2007.08220 (2020). arXiv:2007.08220 https://arxiv.org/abs/2007.08220

[8] Pete Hodgson. 2017. *Feature Toggles (aka Feature Flags)*. https://martinfowler.com/articles/feature-toggles.html

[9] M. Jackson and P. Zave. 1998. Distributed feature composition: a virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering* 24, 10 (1998), 831–847. https://doi.org/10.1109/32.729683

[10] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Larissa Braz, Christian Kästner, Sven Apel, and Kleber Santos. 2020. An Empirical Study on Configuration-Related Code Weaknesses. In *Proceedings of the 34th Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 193–202. https://doi.org/10.1145/3422392.3422409

[11] Jens Meinicke, Juan Hoyos, Bogdan Vasilescu, and Christian Kästner. 2020. Capture the Feature Flag: Detecting Feature Flags in Open-Source. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 169–173. https://doi.org/10.1145/3379597.3387463

[12] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. 2020. Exploring Differences and Commonalities between Feature Flags and Configuration Options. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (Seoul, South Korea) *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 233–242. https://doi.org/10.1145/3377813.3381366

[13] Steve Neely and Steve Stolt. 2013. Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy). *2013 Agile Conference* (2013), 121–128. https://doi.org/10.1109/AGILE.2013.17

[14] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature Toggles: Practitioner Practices and a Case Study. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/2901739.2901745

[15] Doug Seven. 2014. Knightmare: A DevOps Cautionary Tale. https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/

[16] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/2815400.2815401

[17] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. 2021. Static detection of silent misconfigurations with deep interaction analysis. *PACM on Programming Languages* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485517