Feature-Interaction Aware Configuration Prioritization for Configurable Code

Son Nguyen*, Hoan Nguyen†, Ngoc Tran*, Hieu Tran*, and Tien N. Nguyen*
*Computer Science Department, The University of Texas at Dallas, USA,
*Email: {sonnguyen,nmt140230,trunghieu.tran,tien.n.nguyen}@utdallas.edu
†Amazon Corporation, USA, Email: nguyenanhhoan@gmail.com

***Abstract*—Unexpected interactions among features induce most bugs in a configurable software system. Exhaustively analyzing all the exponential number of possible configurations is prohibitively costly. Thus, various sampling techniques have been proposed to systematically narrow down the exponential number of legal configurations to be analyzed. Since analyzing all selected configurations can require a huge amount of effort, fault-based configuration prioritization, that helps detect faults earlier, can yield practical benefits in quality assurance. In this paper, we propose CoPro, a novel formulation of feature-interaction bugs via common program entities enabled/disabled by the features. Leveraging from that, we develop an efficient feature-interaction-aware configuration prioritization technique for a configurable system by ranking the configurations according to their total number of potential bugs. We conducted several experiments to evaluate CoPro on the ability to detect configuration-related bugs in a public benchmark. We found that CoPro outperforms the state-of-the-art configuration prioritization techniques when we add them on advanced sampling algorithms. In 78% of the cases, CoPro ranks the buggy configurations at the top 3 positions in the resulting list. Interestingly, CoPro is able to detect 17 not-yet-discovered feature-interaction bugs.**

***Keywords*-Configurable Code, Feature Interaction; Configuration Prioritization; Software Product Lines;**

I. Introduction

Several software systems enable developers to configure to different environments and requirements. In practice, a highly-configurable system can tailor its functional and non-functional properties to the needs and requirements of users. It does so via a very large number of *configuration options* [9], [10] that are used to control different *features* [5], [29]. For example, Linux Kernel supports more than 12,000 compile-time configuration options, that can be configured to generate specific kernel *variants* for billions of scenarios.

In a configurable system, features can interact with one another in a non-trivial manner. As a consequence, such interaction could inadvertently modify or influence the functionality of one another [59]. Unexpected interactions might induce bugs. In fact, most configuration-related bugs are caused by interactions among features [2], [25], [42], [46], [56]. Unfortunately, traditional methods cannot be directly applied to work on configurable code since they focus on detecting bugs in a particular variant. Furthermore, exhaustively analyzing the systems is infeasible due to the exponential number of all possible configurations. In practice, configuration testing is often performed in a manual and ad-hoc manner by unsystematically selecting common variants for analysis [26], [39].

To systematically perform quality assurance (QA) for a highly-configurable system (Figure 1), researchers have proposed several techniques to narrow the configuration space *by* **eliminating invalid configurations** *that violate the feature model* of the system, which defines the feasible configurations via the constraints among the features [17], [18], [27], [30], [29], [50]. However, the number of configurations that need to be tested is still exponential. To address this explosion problem, researchers introduce various **configuration selection** strategies. The popular strategies include the sampling algorithms which achieve feature interaction coverage such as *combinatorial interaction testing* [48], [47], [28], [40], *one-enabled* [42], *one-disabled* [2], *most-enabled-disabled* [52], *statement-coverage* [53], to reduce the number of configurations to be analyzed. Still, *those algorithms assume the chances of detecting interaction bugs are the same for all those combinations*. Thus, interaction faults might be discovered only after the last variants in such samples is tested. Thus, after configuration selection, the selected *set of configurations need to be* **prioritized** for QA activities [3]. Note that configuration prioritization is different from test case prioritization because after configuration prioritization, any QA activities can be applied on the ranked list of prioritized configurations, including test generation and testing, static bug detection, or manual code review (Figure 1).

To motivate configuration prioritization, let us take an example of Linux Kernel. In Linux, the number of different configuration options is over 12,000, leading to +2^{12K} different configurations. After applying all the constraints on various combinations of options, the number of valid configurations for QA is an exponential number. For configuration selection, by using *six-wise* sampling algorithm, the number is still extremely large, up to +500K configurations [42]. Hence, without configuration prioritization, many bugs that are dependent on configurations might still be hidden due to this large configuration space, especially when the resources for QA (*e.g.,* time and developers' efforts) are limited.

In practice, developers even do not perform QA activities on a particular configuration until it was reported to have defects by the users. In this case, users have already suffered the consequences of those defects. Due to the large number of configurations after selection for QA, even *compile-time*

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00053

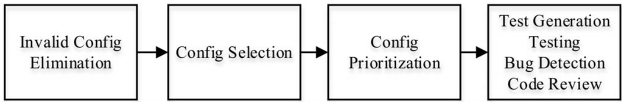

Figure 1. The QA Process of Configurable System

errors and flaws cannot be quickly detected by a compiler or a bug detection in the appropriate configuration. Indeed, in the Variability Bugs Database (VBDb) [2], a public database of real-world configuration-related bugs reported for the Linux kernel, *there are 42 out of 98 bugs and flaws that are compile-time: 25 declaration errors, 10 type errors, and 7 cases of dead code*.

Despite the importance of configuration prioritization, the state-of-the-art methods for such prioritization are still limited in detecting feature-interaction bugs. The similarity-based prioritization method (SP) [3] is based on the idea that dissimilar test sets are likely to detect more defects than similar ones [3]. In SP, the configuration with the maximum number of features is selected to be the first one under test. The next configuration under test is the configuration with the lowest feature similarity compared to the previously chosen one. Despite its success, there are two key problems with SP. First, SP aims to cover as many features different from the previous ones. The different features to be considered next might not be the ones that potentially causes violations. SP does not examine the interaction between features, which is the key aspect causing interaction bugs in a variant. Second, in SP, the quality of the resulted prioritization order strongly depends on the selection of the first configuration.

In this paper, we propose CoPro, a novel configuration prioritization approach for configurable systems by analyzing their code to detect feature-interaction bugs. Our key idea in CoPro is as follows. In a configurable system, features are implemented as blocks of code, which are expressed via the *program instructions/operations (e.g., declarations, references, assignments, etc.) on the data structures/program entities (e.g., variables, functions, etc.)*. Features interaction occurs when the operations on the program entities shared between the features have impacts on each other.

Those operations, when the features are enabled or disabled, potentially create a violation(s) that makes the program not-compilable or having a run-time error. Detecting feature interactions via operations would help identify potential feature-interaction bugs. An example of a violation is that a feature disables the only initialization of a variable while another enables one of its dereferences (the violation of *"dereferencing an un-initialized variable"*). This violation could lead to a `NULL` pointer exception. It is clear that the configuration in which the former feature is disabled and the latter is enabled, is more suspicious than the one where both of them are either enabled or disabled. The suspiciousness of a configuration is indicated via the potential feature-interaction violations. Hence, a higher number of potential violations makes the configuration more suspicious. The suspiciousness

```
#ifdef CONFIG_TWL4030_CORE
#define CONFIG_IRQ_DOMAIN
#endif
#if !defined(CONFIG_SPARC)
int of_platform_populate(int node){
    return 0;                                        L
}
#endif
#ifdef CONFIG_IRQ_DOMAIN
int irq_domain_simple_ops = 1;
void irq_domain_add(int *ops){                       X
    int irq = *ops;
}
#endif
#ifdef CONFIG_TWL4030_CORE
int twl_probe(int n){
    int *ops = NULL, status = n;                     K
#ifdef CONFIG_OF_IRQ
    ops = &irq_domain_simple_ops;                Y
    status = 0;
#endif
    irq_domain_add(ops);
#ifdef CONFIG_OF_DEVICE
    status = of_platform_populate(n);            Z
#endif
    return status;
}
#endif
```

Figure 2. A Simplified Bug in Linux Kernel

levels are used to rank the configurations, which helps testing, bug detection, or other QA activities more efficient.

We conducted several experiments to evaluate CoPro in two complementary settings. First, in a benchmark setting, we ran CoPro on the Variability Bugs Database (VBDb) [2]. We compared CoPro with the two state-of-the-art approaches in *random prioritization* and *similarity-based prioritization* (SP) [3], when we added each of the compared techniques on top of several state-of-the-art sampling algorithms [42]. We found that CoPro significantly outperforms the other techniques. In 78.0% of the cases, CoPro ranks the buggy configurations at the top-3 positions in the list, while the SP and $Random$ approaches rank them at the top-3 positions for only 41.3% and 26.1% of the cases. Interestingly, CoPro was able to detect *17 feature-interaction bugs that were not yet discovered in VBDb* including high-degree interaction bugs, memory leaking bugs, *etc.* In the second setting, we connect CoPro with a compiler to run on large, open-source configurable systems, and CoPro can detect 4 newly discovered bugs and programming flaws.

In summary, in this paper, our main contributions include:

- A formulation of feature-interaction bugs using common program entities enabled/disabled by the features;
- CoPro: an efficient feature-interaction-aware configuration prioritization technique for configurable systems;
- An extensive experimental evaluation showing the effectiveness of CoPro over the state-of-the-art approaches.

II. Motivating Example

In this section, we illustrate the challenges of configuration prioritization and motivate our solution via an example.

A. Examples of Bugs in Configurable Code

Let us consider the simplified version of the real buggy code in the Linux kernel [2] at the commit 40410715715 of Linux-stable at https://git.kernel.org (shown in Figure 2). This version has more than 5,200 compile-time configuration options and about 30,000 files. The code in Figure 2 contains two feature-interaction bugs that were discovered in certain configurations:

- A compile-time error occurs (*use the undeclared function* `of_platform_populate` on line 24) in the variants where `CONFIG_TWL4030_CORE, CONFIG_OF_DEVICE, and CONFIG_SPARC` are enabled.
- A run-time error occurs (*dereferencing the* `NULL` *pointer* `ops` on line 12) in the configurations where `CONFIG_TWL4030-_CORE, CONFIG_OF_DEVICE` are enabled, and `CONFIG_SPARC` and `CONFIG_OF_IRQ` are disabled.

For this example in Linux kernel, brute-force testing of all possible variants to discover these interaction bugs faces the problem of combinatorial explosion in the exponential configuration space (up to $2^{5,200}$ possible configurations). With a huge number of configurations and without an assessment of the potential buggy level of those configurations, the QA process (*e.g.,* debugging) will require a great deal of effort from developers. To deal with such large number of configurations, first, one will eliminate the invalid configurations that violate the *constraints among the features* in the system [17], [18], [27], [29], [30], [50]. However, the number of configurations after this step is still exponential. To balance between bug detection rate and the number of configurations to be examined, the *configuration selection* process is applied. An example of selection algorithms is the *k-way* combinatorial approach [28], [40], [47], [48], which considers the system under test as a blackbox and selects a subset with at most k features. However, even with a small value of k, *e.g.,* $k = 6$, inspecting a very large number of selected configurations without prioritizing the variants most likely having defects is still inefficient. Therefore, one would need a prioritization strategy to rank the configurations to be examined.

The current state-of-the-art configuration prioritization algorithm is the similarity-based configuration prioritization (SP) [3]. Unfortunately, SP is still ineffective in detecting feature-interaction bugs. Let us illustrate this via our example. Table I shows the partial set of configurations chosen by 4-wise sampling algorithm and prioritized by SP [3]. The variant, where `TWL4030_CORE`, `IRQ_DOMAIN`, `OF_IRQ`, and `OF_DEVICE` are enabled, and `SPARC` is disabled, with the maximum number of features is selected to be examined first by the SP algorithm. For the next configuration, the configuration that has the minimum number of similar features compared to the previously selected configuration is picked (*i.e.,* the one in which `TWL4030_CORE`, `IRQ_DOMAIN`, `OF_IRQ`, and `OF_DEVICE` are disabled, and `SPARC` is enabled). Although this second configuration is most dissimilar to the first one, it does not contain the features whose interactions cause violations, and there is no bug revealed by the second configuration. As a result, by SP's strategy, the result is not an efficient

Table I
THE CONFIGURATIONS ORDERED BY SP ALGORITHM [3] FOR FIGURE 2

#	OF_IRQ	IRQ_DOMAIN	OF_DEVICE	TWL4030_CORE	SPARC
1	T	T	T	T	F
2	F	F	F	F	T
3	F	T	F	F	T
4	**F**	**T**	**T**	**T**	**F**
5	T	T	F	T	T
6	**T**	**T**	**T**	**T**	**T**
7	**F**	**T**	**T**	**T**	**T**

order for inspection because the aforementioned compile-time and run-time errors are not detected until the 4^{th} and 6^{th} configurations are inspected respectively. The configuration with both interaction bugs would only be discovered via the 7^{th} configuration. In our experiment (will be presented in Section V), 36.2% of the feature-interaction bugs in the public benchmark, the Variability Bugs Database (VBDb) [2], cannot be revealed until at least 10 configurations are inspected in the resulting list ranked by the SP approach.

B. Observations

Let us consider the code in Figure 2 with the two following feature interactions that can cause the violations of program semantics: 1) the declaration of the function `of_platform_populate` in feature `L` (line 5) and its use in `Z` (line 24), and 2) the assignment of `ops` in feature `Y` (line 19) and its reference in `K` (line 22). There are two potential bugs: 1) the use of the function `of_platform_populate` without its declaration; and 2) the reference to the variable `ops` without its initiation. The configuration that enables `Z` and disables `L` (`CONFIG_OF_DEVICE=T`, `CONFIG_SPARC=T`) and enables `K` and disables `Y` (`CONFIG_TWL4030_CORE=T`, `CONFIG_OF_IRQ=F`) should be inspected earlier to detect the two bugs. Based on this observation, those interactions between features should be comprehended to quickly discover these above interaction bugs. That motivates us to propose an approach that first analyzes the source code to more precisely detect the potential interactions among features, and then assesses the probabilities to be faulty of the configurations to prioritize to inspect/test them in a more efficient order.

C. COPRO *Overview*

Detecting all interactions among features in a sound and complete manner requires an analysis on all combinations of configuration options. That is prohibitively expensive and impractical. To deal with this problem, we statically analyze the source code to *locally and heuristically identify the interactions between features via the shared program entities and the operations on them*. For example, `L` shares the function `of_platform_populate` with `Z` (which is declared on line 5 and used on line 24) and `K` interacts with `Y` via the variable `ops` (which is assigned on line 19 and referred to on line 22). Importantly, the operations such as declaration, assignment, or references on the shared entities could become invalid when certain features (via configuration options) are enabled or disabled. As a consequence, that could lead to a violation. For instance, a violation occurs when `CONFIG_TWL4030_CORE`, `CONFIG_OF_DEVICE`, and `CONFIG_SPARC` are enabled because the function `of_platform_populate` would be used (`K` and `Z` are enabled) while its declaration is turned off (`L` is disabled). The other violation occurs in the case that `CONFIG_TWL4030_-CORE` is `true`, that enables `K`, while `Y` is disabled as `CONFIG_OF_IRQ` is disabled. This would induce the bug of *dereferencing the* `NULL` *pointer* on variable `ops` (line 12). With our strategy, the 7^{th} variant in Table I is more suspicious than the 4^{th}, 6^{th}, and any other ones. Generally, the suspiciousness of a variant is determined by the number of violations that it might induce. Finally, a configuration can be ranked according to its suspiciousness score, thus, we could create a prioritization order of variants that maximizes fault detection rate.

III. Formulation

Let us formulate the problem of feature-interaction-aware configuration prioritization.

A. Program Entities and Operations

In a program, we are interested in the program entities and the operations performed on them.

Definition 1. *(***Program Entity***). A program entity is a program element that is uniquely identified with a name and a scope. The scope and the name of an entity are used together as the identity of the entity.*

In our formulation, we are interested in two types of program entities: *variable* and *function*. An entity is represented in the form of `[scope.ent_name]`, where `scope` and `ent_name` are the scope and the name of the program entity respectively. For example, the code in Figure 2 contains the variables `GLOBAL.irq_domain_simple_ops`, `twl_probe.ops`, the function `GLOBAL.twl_probe`, *etc.*

We define 4 types of operations on variables and functions.

Definition 2. *(***Program Operation***). We define four types of operations on variables and functions: declare, assign, use and destruct. Let OP be the set of program operations, $OP = \{declare, assign, use, destruct\}$. All of those four operations are applicable to variables, while $declare$ and use are only applicable on functions.*

For variables, the *assign* operation is used to assign a non-null value to a variable. A `NULL` assignment to a variable is treated as a special case of an assignment. In Figure 2, function `GLOBAL.of_platform_populate_probe` is *declared* at line 5, and *used* at line 24. `twl_probe.ops` is *declared* (line 17), *assigned* a value (line 19), and then *used/referred* to (line 22).

B. Configurations and Features

A configurable system contains several segments of code that are present in any variant that implements its basic functionality. Those segment form *the core* of the system.

In practice, a configurable system usually provides a large number of **configuration options** to configure several optional segments of code to be present or absent, in addition to the core of the system. Those optional segments of code are aimed to realize the optional **features** of the system. For example, in the Linux Kernel, the configuration options have the prefix of `CONFIG_`, and they can have different values. Without loss of generality, we assume that the value of a configuration option is either `true(T)` or `false(F)` (We can consider the entire conditional expressions of non-boolean options as boolean ones, *e.g.,* `CONFIG_A>10` as `CONFIG_A>10=T/F`).

Definition 3. *(***Configuration Option***). A configuration option (*option *for short) is an element that is used to configure the source code of a configurable system, such that the option's value determines the presence or absence of one or more segments of code.*

In a configurable system, the presence or absence of code segments is dependent on the values of multiple options. In Figure 2, the lines 19 and 20 are presented only when both `CONFIG_TWL4030_CORE` and `CONFIG_OF_IRQ` are `T`. Thus, at line 19, `irq_domain_simple_ops` is potentially used to assign as a value to the variable `ops` when both of those options are `T`.

Definition 4. *(***Selection Functions***). In a configurable system, we define selection functions as the functions from $O \times V$ to 2^P, where O is the set of configuration options, $V = \{\texttt{T, F}\}$, and P is the set of program entities used in the code of the configurable system. We define four selection functions:*

- *$\alpha : O \times V \to 2^P$, $\alpha(o, v) = D$, where $o \in O, v \in \{\texttt{T, F}\}$, and D is the set of entities potentially* **declared** *if $o = v$.*
- *$\beta : O \times V \to 2^P$, $\beta(o, v) = D$, where $o \in O, v \in \{\texttt{T, F}\}$, and D is the set of entities potentially* **assigned** *if $o = v$.*
- *$\gamma : O \times V \to 2^P$, $\gamma(o, v) = D$, where $o \in O, v \in \{\texttt{T, F}\}$, and D is the set of entities potentially* **used** *if $o = v$.*
- *$\delta : O \times V \to 2^P$, $\delta(o, v) = D$, where $o \in O, v \in \{\texttt{T, F}\}$, and D is the set of entities potentially* **destructed** *if $o = v$.*

For example, in Figure 2:

- α(`CONFIG_SPARC, F`)={`GLOBAL.of_platform_populate, of_platform_populate.node`}
- β(`CONFIG_OF_IRQ, T`)={`twl_probe.ops`}
- γ(`CONFIG_OF_IRQ, T`)={`GLOBAL.irq_domain_simple_ops`}

Definition 5. *(***Configuration***). Given a configurable system, a configuration is a specific selection of configuration options, which defines a* ***variant*** *of the system.*

Configuration options are used to control the features that are represented by certain segments of code. For example, in Figure 2, the feature represented by the segment of code `X` (feature `X`) is enabled if the value of the configuration option `CONFIG_IRQ_DOMAIN` is `true`, whereas feature `Y` is enabled if both `CONFIG_OF_IRQ` and `CONFIG_TWL4030_CORE` are `true`.

Definition 6. *(***Feature***). In a configurable system, a feature f is implemented by applying program operations on a set of program entities, whose presence/absence is controlled by certain configuration options. We denote it by $f \sim OP \times \rho$*

Table II
DIFFERENT KINDS OF FEATURE INTERACTIONS

	Kind of Interaction	Condition
1	*declare-declare*	$\exists e \in \rho_1 \cap \rho_2$, e is declared in both f_1 and f_2
2	*declare-assign*	$\exists e \in \rho_1 \cap \rho_2$, e is declared in f_1 and then assigned in f_2
3	*declare-use*	$\exists e \in \rho_1 \cap \rho_2$, e is declared in f_1 and used in f_2
4	*declare-destruct*	$\exists e \in \rho_1 \cap \rho_2$, e is declared in f_1, and destructed in f_2
5	*assign-assign*	$\exists e \in \rho_1 \cap \rho_2$, e is assigned in both f_1 and f_2
6	*assign-use*	$\exists e \in \rho_1 \cap \rho_2$, e is assigned in f_1 and used in f_2
7	*assign-destruct*	$\exists e \in \rho_1 \cap \rho_2$, e is assign in f_1 and destructed in f_2
8	*use-destruct*	$\exists e \in \rho_1 \cap \rho_2$, e is used in f_1 and destructed in f_2
9	*destruct-destruct*	$\exists e \in \rho_1 \cap \rho_2$, the entity is destructed in both f_1 and f_2

where OP is the set of program operations and ρ is the set of program entities.

A special case of features is that f is *the core feature* (F), $A \cup B \cup \Gamma \cup \Delta = \rho$, where A, B, Γ, Δ are the sets of program entities that are declared, assigned, used and destructed in the core system. F is not controlled by any configuration option.

C. Feature Interactions

In a configurable system, a feature may influence or modify (often called *interact* with) the functions offered by other features through shared program entities that are used to implement the features. For example, features `X`, `K` and `Z` interact with one another via the variables `GLOBAL.irq_domain_simple_ops` and `twl_probe.temp`. The manners the features interacting with each other depend on how the shared entities are operated. For example, feature `Y` *assigns* `&irq_domain_simple_ops` to `ops` and feature `K` *uses* that variable (line 22). If no assignment was done in `Y`, dereferencing in `K` would be invalid, causing a `NULL` pointer exception.

Multi-way feature-interaction. We present only on the interactions between pairs of features because the interactions between more than two features can be modeled as the operations on the shared variables between pairs of features. Let us provide a sketch of the proof for this statement. We assume that there exists an interaction among m features ($m > 2$). For simplicity, we consider the case of $m = 3$, and the interaction among $f_1 \sim OP \times \rho_1, f_2 \sim OP \times \rho_2$ and $f_3 \sim OP \times \rho_3$. There are two cases of this interaction. First, there exists an entity that shared by all 3 features, $\rho_1 \cap \rho_2 \cap \rho_3 = \omega \neq \emptyset$. Since $\rho_1 \cap \rho_2 \supset \omega$ and $\rho_2 \cap \rho_3 \supset \omega$, identifying interactions between pairs directly captures the interaction among 3 features. The second case is that $\rho_1 \cap \rho_2 = \omega_1$, $\rho_2 \cap \rho_3 = \omega_2$ and $\omega_1 \cap \omega_2 = \emptyset$. Meanwhile, f_3 is influenced by f_1 (because the roles of f_1 and f_3 features in this case are equal). This leads to that there exist entities: $e_1 \in \omega_1, e_2 \in \omega_2$, such that $e_2 = p(e_1)$, where p is a value propagation function. This means the value of e_1 is propagated to e_2, and that influences f_3. Hence, the interaction among 3 features is still captured by determining interactions between pairs of features.

For instance, the interaction among features `X`, `K` and `Z` can be broken down into the shared program entities between two pairs of features as follows: `(X, K)` via the variable `GLOBAL.irq_domain_simple_ops`, and `(K, Z)` via the variable `twl_probe.temp`. Thus, our solution can still model the interactions with more than two features via the operations on their shared program entities. From now on, we refer to a feature interaction as an interaction determined via the shared program entities between a pair of features.

In COPRO, we focus on the feature interaction through the shared program entities. The feature interactions when the variables are associated with the external data such as when they interfere with each other's behaviors on files or databases are beyond the scope of our static analysis-based solution. Similarly, we will not detect the interactions through pointers or arrays in this work. As a consequence, if both features *use* (refer to) a program entity, they will not change the program's state. Thus, there is no interaction between two features if they only use shared functions and variables.

With the above design focuses, in COPRO, the interactions between two features $f_1 \sim OP \times \rho_1$, and $f_2 \sim OP \times \rho_2$ with $\rho_1 \cap \rho_2 \neq \emptyset$, can be categorized into nine kinds of interactions that are displayed in Table II (the *use-use* case is eliminated as explained).

D. Feature Interaction Detection

In a configurable system, the features (except the core features of the system) are controlled by certain configuration options. Thus, if there exists an interaction among the features, the interaction will be one of the following:

- *declare-declare*, there exist two option o_1, o_2 and their selected values v_1, v_2, such that $\alpha(o_1, v_1) \cap \alpha(o_2, v_2) \neq \emptyset$
- *declare-assign*, there exist two option o_1, o_2 and their selected values v_1, v_2, such that $\alpha(o_1, v_1) \cap \beta(o_2, v_2) \neq \emptyset$
- *declare-use*, there exist two option o_1, o_2 and their selected values v_1, v_2, such that $\alpha(o_1, v_1) \cap \gamma(o_2, v_2) \neq \emptyset$
- *declare-destruct*, there exist two option o_1, o_2 and their selected values v_1, v_2, such that $\alpha(o_1, v_1) \cap \delta(o_2, v_2) \neq \emptyset$
- *assign-assign*, there exist two option o_1, o_2 and their selected values v_1, v_2, such that $\beta(o_1, v_1) \cap \beta(o_2, v_2) \neq \emptyset$
- *assign-use*, there exist two option o_1, o_2 and their selected values v_1, v_2, such that $\beta(o_1, v_1) \cap \gamma(o_2, v_2) \neq \emptyset$
- *assign-destruct*, there exist two option o_1, o_2 and their selected values v_1, v_2, such that $\beta(o_1, v_1) \cap \delta(o_2, v_2) \neq \emptyset$
- *use-destruct*, there exist two option o_1, o_2 and their selected values v_1, v_2, such that $\gamma(o_1, v_1) \cap \delta(o_2, v_2) \neq \emptyset$
- *destruct-destruct*, there exist two option o_1, o_2 and their selected values v_1, v_2, such that $\delta(o_1, v_1) \cap \delta(o_2, v_2) \neq \emptyset$

Based on the above rules, our feature-interaction detection algorithm statically analyzes the source code and configuration options, and then computes the sets α, β, γ, and δ for any two options o_1 and o_2. For example, we can detect a *declare-declare* interaction between 2 features if there exists 2 options o_1 and o_2, such that $\alpha(o_1, v_1) \cap \alpha(o_2, v_2) \neq \emptyset$, where v_1, v_2 are their selected values. Other detection rules are similarly derived. For example, because $\beta($`CONFIG_OF_IRQ, T`$)\cap\gamma($`CONFIG_TWL4030_CORE, T`$) = \{$`ops`$\}$, there is a potential *assign-use* interaction among features. Thus, in this case, the actual *assign-use* interaction among `Y` and `K` exists.

For the core feature, if F and other features interact with one another, depending on the kinds of the interaction, there exists a selection v of an option o, such that $\alpha(o, v), \beta(o, v), \gamma(o, v), \delta(o, v)$ intersect with A, B, Γ, Δ, *i.e.,* intersecting with the entities in the core. Interactions among core features and others are similarly identified.

In this version of CoPro, we formulate feature interaction statically through the completed set of operations on the entities that are shared between features. More sophisticated interactions relevant to pointers and external data such as files or databases can be detected by using different data structures in the same principle and using other types of analysis.

IV. Configuration Prioritization

A. Overview

In general, to prioritize a given set of configurations under test, our algorithm assigns a suspiciousness score to each configuration. The suspiciousness score is determined via the number of the potential feature-interaction bugs in different kinds that the variant corresponding to that configuration might potentially have.

Feature-interaction bugs can be induced by any kinds of interaction. Table III shows 10 different kinds of feature-interaction bugs that are potentially caused by the respective kinds of interactions listed in Table II of Section III. The interactions in Table III are called *sensitive interactions* with their *suspicious selection* of options. A configuration containing a suspicious selection potentially has the corresponding violation. For example, at line 6, if $\beta(o_1, v_1) \cap \gamma(o_2, v_2) \neq \emptyset$, there is an *assignment-use* potential interaction between f_1 and f_2. When $o_1 = v_1', o_2 = v_2$, where $v_1' \neq v_1$, f_1 might be disabled while f_2 is enabled, which poses a violation of *use without assignment*. In Figure 2, because $\alpha($`CONFIG_SPARC, F`$) \cap \gamma($`CONFIG_OF_DEVICE,T`$) = \{$`GLOBAL.of_platform_populate`$\}$, the program might not be compiled if `CONFIG_SPARC = T` and `CONFIG_OF _DEVICE = T` (*use without declaration*).

B. Detailed Algorithm

The listing 1 shows the pseudo-code of CoPro, our feature-interaction aware configuration prioritization algorithm. Given a configurable system, we first extract the set of options used in the system. Then, for each selection v of each option o, the sets $\alpha(o, v), \beta(o, v), \gamma(o, v)$, and $\delta(o, v)$ are computed via the function $CollectProgramEntities$ (lines 4–5). After that, for

Algorithm 1 CoPro: Feature-Interaction aware Configuration Prioritization Algorithm

1: **procedure** DetectSuspiciousSelections($Code$)
2: $Options = ExtractOptions(Code)$
3: **for all** $o \in Options$ **do**
4: $TSelc = CollectProgramEntities(o, T, Code)$
5: $FSelc = CollectProgramEntities(o, F, Code)$
6: $Selections.add(TSelc)$
7: $Selections.add(FSelc)$
8: **for all** $selc \in Selections$ **do**
9: **for all** $other \in Selections$ **do**
10: **if** $ExistInteraction(selc, other)$ **then**
11: **if** $IsSensitiveInteraction(selc, other)$ **then**
12: $ss = ExtractSuspSelection(selc, other)$
13: $SuspiciousSelections.add(ss)$

14: **procedure** Prioritize($Configurations$, $SuspSelections$)
15: **for all** $c \in Configurations$ **do**
16: $SScore = CaculateSuspScore(c, SuspSelections)$
17: $SetScore(c, SScore)$
18: $OrderBySuspiciousnessScoreDesc(Configurations)$

each pair of option selections, it detects the potential interactions among the features and checks whether the interactions are sensitive as described in Table III. Sensitive interactions are used to specify suspicious selections. This information is used to compute the suspiciousness score for each configuration after configuration selection (line 16). This score is the number of suspicious selections contained by a configuration, and equal to the number of potential bugs that the corresponding variant might have. Finally, the configurations are ranked descendingly by their suspiciousness scores.

C. Static Analysis

In this version of CoPro, to compute α, β, γ, and δ for the value v of an option o in $CollectProgramEntities$, CoPro analyzes the code by using *TypeChef*, a variability-aware parser [33]. For a given configurable code, *TypeChef* is used to analyze and generate the corresponding variability-aware control-flow graph. In a variability-aware control-flow graph, the nodes refer to statements and the edges, which are annotated with the corresponding presence conditions, refer to the possible successor statements (conditional statements). For the example in Figure 2, the successor of the statement at line 22 is the conditional statement at line 24 if `CONFIG_OF_DEVICE` is on, otherwise the statement at line 26 is the direct successor of the statement at line 22. After that, CoPro analyzes every conditional statements in the generated control-flow graph to identify the entities that are either declared, defined, used, or destructed in the statement and compute α, β, γ, and δ for the options and its values in the corresponding presence conditions. For the statement at line 24 in Figure 2, if the value of `CONFIG_OF_DEVICE` is `T`, the variable `status` is defined by using `of_platform_populate` and `n`. This leads to that the variable `status` is in $\beta($`CONFIG_OF_DEVICE,T`$)$, and $\gamma($`CONFIG_OF_DEVICE,T`$)$ contains the function `of_platform_populate` and the variable `n`.

Table III
DIFFERENT KINDS OF FEATURE-INTERACTION DEFECTS

	Kind of interaction	Detection condition	Suspicious selection	Potential violation
1	*declare-declare*	$\alpha(o_1, v_1) \cap \alpha(o_2, v_2) \neq \emptyset$	$o_1 = v_1, o_2 = v_2$	Declaration duplication
2	*declare-use*	$\alpha(o_1, v_1) \cap \gamma(o_2, v_2) \neq \emptyset$	$o_1 = v_1', o_2 = v_2$	Use without declaration
3	*declare-use*	$\alpha(o_1, v_1) \cap \gamma(o_2, v_2) \neq \emptyset$	$o_1 = v_1, o_2 = v_2'$	Unused variables/functions
4	*declare-destruct*	$\alpha(o_1, v_1) \cap \delta(o_2, v_2) \neq \emptyset$	$o_1 = v_1', o_2 = v_2$	Destruction without declaration
5	*declare-assign*	$\beta(o_1, v_1) \cap \beta(o_2, v_2) \neq \emptyset$	$o_1 = v_1, o_2 = v_2$	Assignment without declaration
6	*assign-use*	$\beta(o_1, v_1) \cap \gamma(o_2, v_2) \neq \emptyset$	$o_1 = v_1', o_2 = v_2$	Use without assignment
7	*assign-destruct*	$\beta(o_1, v_1) \cap \delta(o_2, v_2) \neq \emptyset$	$o_1 = v_1', o_2 = v_2$	Destruction without definition
8	*assign-destruct*	$\beta(o_1, v_1) \cap \delta(o_2, v_2) \neq \emptyset$	$o_1 = v_1, o_2 = v_2'$	Memory leak
9	*destruct-destruct*	$\delta(o_1, v_1) \cap \delta(o_2, v_2) \neq \emptyset$	$o_1 = v_1, o_2 = v_2$	Destruction duplication
10	*destruct-use*	$\delta(o_1, v_1) \cap \gamma(o_2, v_2) \neq \emptyset$	$o_1 = v_1, o_2 = v_2$	Use after destruction

Table IV
CONFIGURATION OPTIONS AND THE VALUES OF 4 SELECTION FUNCTIONS α, β, γ, AND δ FOR THE EXAMPLE IN FIGURE 2

Option	Value	α	β	γ	δ
`OF_IRQ`	T		`twl_probe.ops`	`GLOBAL.irq_domain_simple_ops`	
`IRQ_DOMAIN`	T	`GLOBAL.irq_domain_simple_ops,` `GLOBAL.irq_domain_add,` `irq_domain_add.irq,` `irq_domain_add.ops`	`irq_domain_add.irq`	`irq_domain_add.ops`	
`OF_DEVICE`	T			`GLOBAL.of_platform_populate`	
`SPARC`	F	`GLOBAL.of_platform_populate,` `of_platform_populate.node,` `of_platform_populate.t`			
`TWL4030_CORE`	T	`GLOBAL.twl_probe, twl_probe.n,` `twl_probe.status, twl_probe.temp,` `twl_probe.ops`	`twl_probe.node,` `twl_probe.temp,` `twl_probe.status,` `twl_probe.ops`	`GLOBAL.irq_domain_simple_ops,` `GLOBAL.of_platform_populate,` `GLOBAL.irq_domain_add,` `twl_probe.node, twl_probe.temp,` `twl_probe.status, twl_probe.ops`	

Table V
TOP-3 CONFIGURATIONS RANKED BY COPRO FOR FIGURE 2

Rank by CoPro	Rank by SP	`OF_IRQ`	`IRQ_DOMAIN`	`OF_x DEVICE`	`SPARC`	`TWL4030_CORE`	Score
1	7	F	T	T	T	T	3
2	6	T	T	T	T	T	2
3	4	T	F	T	T	F	2

D. Running Example

Let us illustrate our algorithm via the example shown in Figure 2. CoPro computes the sets of the selection functions for each option, and the result is shown in Table IV. Based on the description on Table III, the *suspicious* selections include:

- `CONFIG_OF_IRQ=F`, `CONFIG_TWL4030_CORE = T`
- `CONFIG_SPARC=T`, `CONFIG_TWL4030_CORE = T`
- `CONFIG_SPARC=T`, `CONFIG_OF_DEVICE = T`
- `CONFIG_IRQ_DOMAIN=F`, `CONFIG_OF_IRQ = T`
- `CONFIG_IRQ_DOMAIN=F`, `CONFIG_TWL4030_CORE = T`

Based on the suspicious selections, CoPro assigns the suspiciousness scores and ranks all the configurations accordingly. Table V shows the ranked configurations for our example with their corresponding scores. The top-ranked configuration by CoPro is the 7^{th} configuration in the order generated by the *ACTS* tool [58], a combinatorial test generation tool (see Table I). The configuration covers both interaction bugs. Thus, after inspecting/testing the first configuration, those two bugs will be detected. In other words, CoPro effectively ranks higher the potential buggy variant than the SP algorithm.

V. EMPIRICAL EVALUATION

To evaluate our configuration prioritization approach, we sought to answer the following:

RQ1 **[Performance against a benchmark].** How does CoPro perform on Variability Bugs Database (VBDb) [2], a public dataset of bugs in configurable code?

RQ2 **[Comparison].** How does CoPro improve over the baseline random prioritization and similarity-based prioritization [3] approaches when we add each of them on top of advanced sampling configuration selection algorithms?

RQ3 **[Performance in the wild].** How does it perform on not-yet discovered interaction bugs in configurable systems?

RQ4 **[Time Complexity]** What is CoPro's running time?

To answer RQ1 and RQ2, we conducted an experiment to evaluate CoPro in a controlled environment with the VBDb public benchmark of configuration-related bugs [2]. Answering RQ2 helps evaluate how much improvement CoPro gains over the *random prioritization* and the state-of-the-art *similarity-based prioritization* [3], when adding CoPro on top of the advanced configuration selection techniques [42]. We answer RQ3 to evaluate CoPro in the real-world setting. While the bug detection tools cannot directly work on configurable code, with CoPro, we run them on the list of suspicious configurations ranked by CoPro.

A. Subject Systems

To evaluate CoPro, we used two datasets in two different experiments. To answer RQ1 and RQ2, we used the

Table VI
SUBJECT SYSTEMS IN VARIABILITY BUGS DATABASE

Systems	MinOpt	MaxOpt	MinFile	MaxFile	#Bugs
Linux	3463	5504	18886	34012	43
Busybox	349	1449	236	799	18
HTTPD	602	791	264	426	23
Marlin	243	715	38	135	14

Variability Bugs Database (VBDb) [2] as a benchmark. This publicly available bug database has 98 manually verified configuration-related bugs in different versions of highly-configurable systems: the Linux kernel [38], BusyBox [13], Marlin [41], and Apache HTTPD [4]. Because the VBDb contains configuration-related bugs other than feature-interaction ones, we kept only 46 feature-interaction bugs in those systems. Table VI shows their information including the minimum and maximum numbers of configuration options (MinOpt, MaxOpt), the minimum and maximum numbers of files (MinFile, MaxFile), and the number of feature-interaction bugs (Bugs).

For the second experiment of RQ3, we selected an open-source configurable system with a long history: libpng [36] v0.89 with 40KLOC in 19 files and 80 options, and xterm [57] v2.24 with 50KLOC in 50 files, and 501 configuration options.

B. Experimental Procedure

For each known buggy version of a subject system, we chose to include the maximum number of files of 100 and the maximum number of inclusion level of 3 (due to the limitation of the TypeChef tool [32] that we used for variability-aware parsing). We first applied a configuration selection process. That is, to produce the sampled sets of configurations for each buggy version, we ran sampling algorithms to select a subset of configurations. For each buggy system version and a particular sampling algorithm, we ran COPRO on the set of configurations selected by a sampling algorithm. For comparison, we ran the random prioritization and similarity-based prioritization techniques [3] on the same configurations.

To evaluate COPRO on detecting not-yet reported interaction bugs in VBDb, we first ran it on a subject system to achieve the ranked list of the configurations. We also collected and analyzed the sensitive interactions and potential suspicious selections reported by our tool to detect unknown bugs. For the top-ranked configurations in the list with the reported potential suspicious interactions, we used a compiler to detect bugs.

C. Evaluation Metric

For evaluation, we adopted the *Average Percentage Faults Detected (APFD)* [51], a widely-used metric in evaluating test prioritization techniques. *APFD* is originally applied for evaluating the average percentage bugs detected by a test suite. In this work, since we used COPRO with a bug detection tool, we used *APFD* to measure prioritization effectiveness in term of the rate of bug detection of a configuration set, which is defined by the following formula:

$$APFD = 1 - \frac{\sum_{i=1}^{m} CF_i}{n \times m} + \frac{1}{2 \times n}$$

Table VII
AVERAGE APFD FOR COPRO VERSUS *SP* AND *Random* PRIORITIZATION (ADDED ON TOP OF ADVANCED SAMPLING ALGORITHMS)

	APFD			**AVG Rank**		
	Random	*SP*	COPRO	*Random*	*SP*	COPRO
Pairwise	0.68	0.75	0.93	5.12	4.11	1.55
Three-wise	0.83	0.89	0.96	7.80	4.79	2.39
Four-wise	0.88	0.94	0.97	11.57	6.26	3.77
Five-wise	0.89	0.93	0.97	11.03	6.74	3.49
One-enabled	0.64	0.69	0.91	36.87	30.55	13.19
One-disabled	0.60	0.56	0.88	37.34	38.21	14.76
Most-enabled -disabled	0.52	0.55	0.57	1.70	1.43	1.43
Statement -coverage	0.61	0.57	0.88	37.30	38.25	17.80

where n and m denote the number of configurations and the number of bugs, respectively. CF_i is the smallest number of configurations in the list, which is needed to be inspected to detect the i^{th} bug. The *APFD* score is from 0 to 1. For the fixed numbers of faults and configurations, the higher *APFD*, the higher fault-detection rate and the better ranking order.

D. Effectiveness and Comparison (RQ1 and RQ2)

1) **Comparative Results***:* Table VII shows the comparative results in term of the average APFD and average rank (AVG Rank) between COPRO and the state-of-the-art prioritization methods, when we ran all of them on the results of the advanced sampling techniques [42]. As seen, **COPRO achieves 2–32% higher APFD (14.9% on average) compared to *SP* and 5–28% higher (17.8% on average) compared to *Random* approach**. COPRO also achieves much better ranking compared to *SP* and *Random*. For example, using COPRO with *One-disabled*, which is recommended by the authors of VBDb [2], the interaction bugs are revealed after no more than 15 configurations on average in the resulting ranked list by COPRO are inspected, instead of more than 37 configurations in the lists prioritized by *SP* and *Random*. Especially, in **78.0% of the cases, COPRO ranks the buggy configurations at the top-3 positions in the list, while the *SP* and *Random* approaches rank them at the top-3 positions for only 41.3% and 26.1% of the cases**.

We can also see that COPRO outperforms the *SP* and *Random* prioritization techniques consistently on the resulting configurations selected by various advanced sampling algorithms. That is, if one uses COPRO to rank the configurations selected by advanced algorithms, the inspection order by COPRO is better than those of the *SP* and *Random* prioritization. Note that in the case of *Most-enabled-disabled* [42], for each buggy system, there are only two configurations selected by the sampling algorithm, and 23 out of 46 bugs cannot be revealed by the selected set of configurations. That makes all three prioritization approaches do not perform well in this case and achieve nearly equal average APFDs and ranks. In brief, COPRO *is able to rank the buggy configuration in a much higher rank than SP and Random approaches*. In other words, *if we add* COPRO *as the prioritization technique on top of the most advanced sampling algorithms, we would achieve a more* *effective solution than adding other prioritization approaches upon the selection algorithms.*

2) **Further Analysis***:* We further studied the cases in which CoPro correctly ranks the buggy configurations at the top positions. For the cases with correct ranking (1–3), we found that in 77% (30 out of 39) of these bugs, the features interact with one another via shared program entities. Thus, our rules in Section III are applicable to detect the majority of feature-interaction bugs in the public VBDb benchmark.

We also found an interesting scenario of **indirect feature-interactions** that CoPro detected. In some of those 30 cases, CoPro identifies sensitive interactions among features indirectly via entities. For example, variable `x` is initialized in the feature controlled by option `A` with `A=T`. `x` is assigned to `y` in the feature enabled if the option `B` is on. Then, `y` is referred to in another place that controlled by option `C`, `C=T`. In this case, if `A=F`, `B=T`, and `C=T`, a *null pointer exception* might be induced. In this case, since of the propagation of variables' values, the interaction between two features controlled by `A` and `C` can be captured by CoPro via the feature controlled by `B`. Thus, the buggy configurations are ranked on the top. This also indicates CoPro's capability in detecting configurations containing **bugs relevant to more than two features**.

3) **Examples on Feature-Interaction Bugs***:* Let us present the configuration-related bugs involving **high-degree feature interactions** and the cases that CoPro detected the feature-interaction bugs **not-yet-discovered** in the VBDb benchmark.

A bug involving 6 configuration options. Figure 3 shows a bug in Apache HTTPD at commit 2124ff4. The bug is in the file `mod_cgid.c`. In this example, the bug is observed when `RLIMIT_CPU`, `RLIMIT_NPROC`, `RLIMIT_DATA`, `RLIMIT_VMEM`, and `RLIMIT_AS` are disabled, while `RLIMIT_NPROC` is enabled. With the selections of the combinations of those options, the field `limits` of any variable of the type `cgid_red_t` (e.g. `req`) used in any features is not declared (line 3). Meanwhile, the filed `limits` is used in `req.limits` on line 12 when `RLIMIT_NPROC` is enabled. By identifying the suspicious interactions between the features controlled by the pairs of `RLIMIT_NPROC` and each of these 5 other options via the field `req.limits`, CoPro specifies that the selection that `RLIMIT_NPROC = T`, `RLIMIT_CPU = F`, `RLIMIT_NPROC = F`, `RLIMIT_DATA = F`, `RLIMIT_VMEM = F`, and `RLIMIT_AS = F` is more suspicious than all other selections containing those six configuration options.

Not-yet discovered feature-interaction bugs in VBDb benchmark. Interestingly, while using VBDb, we were able to use CoPro detect the interaction bugs that were neither discovered and reported in those systems nor in VBDb. In total, **we found 17 such feature-interaction bugs including 12 *using-without-declaration* bugs, 2 *memory-leak* bugs, 2 *declaration duplication* bugs, and 1 *dead code* issue**.

Figure 4 shows 2 not-yet-discovered bugs: a *memory leak* issue and an *assignment without declaration* bug at commit fac312d78bf (which also has *use without declaration* bug and *destruction without declaration* bug). The *assignment without declaration* bug occurs only if `BB_FEATURE_LS_SORTFILES = F` and `CONFIG_FEATURE_LS_RECURSIVE = T`. In this case, `dndirs` and `dnd` are not declared since lines 3–4 are not included, but they are used at lines 11 and 13. Moreover, `dnd` is destructed on line 15. This bug was fixed at commit ea224be6aa8 (in almost 6 years later). 3 years after that, a *memory leak* issue was reported and fixed at commit ffd4774ad25: as `CONFIG_FEATURE_LS_RECURSIVE` is disabled, the memory controlled by `subdnp` is initialized at line 9 and not released. With CoPro, it would have been fixed earlier.

A run-time feature-interaction Bug in Busybox CoPro is also able to detect run-time errors caused by feature interactions. Figure 5 shows a simplified bug in Busybox extracted from http://vbdb.itu.dk/#bug/busybox/061fd0a. In this case, a bug occurs when `CONFIG_FEATURE_HDPARM_HDIO_UNREGISTER_HWIF = T` if `c='U'` and `p = NULL`. The execution goes to `expected_hwif_error`. However, this label is visible only when `CONFIG_FEATURE_HDPARM_HDIO_SCAN_HWIF = T`. Otherwise, we would have a run-time error.

```
1  typedef struct {
2  #if defined (RLIMIT_CPU) || defined (RLIMIT_NPROC) ||
       defined (RLIMIT_DATA) || defined(RLIMIT_VMEM ) ||
       defined(RLIMIT_AS)
3    cgid_rlimit_t limits;
4  #endif
5  } cgid_req_t;
6  static apr_status_t send_req(){
7    cgid_req_t req = {0};
8  #if defined(RLIMIT_DATA) || defined(RLIMIT_VMEM) ||
       defined(RLIMIT_AS)
9    req.limits.limit_mem_set = 1;
10 #endif
11 #ifdef RLIMIT_NPROC
12   req.limits.limit_nproc = 0;
13 #endif
14 }
```

Figure 3. A 6-way Feature-Interaction Bug in Apache Httpd

```
1  void showdirs(struct dnode **dn, int ndirs){
2  #ifdef BB_FEATURE_LS_SORTFILES
3    int dndirs;
4    struct dnode **dnd;
5  #endif
6    subdnp = list_dir(dn[i]->fullname);
7  #ifdef CONFIG_FEATURE_LS_RECURSIVE
8    dnd = splitdnarray(subdnp, nfiles);
9    dndirs = countsubdirs(subdnp, nfiles);
10 #ifdef CONFIG_FEATURE_LS_SORTFILES
11   shellsort(dnd, dndirs);
12 #endif
13   showdirs(dnd, dndirs);
14   free(dnd);
15   free(subdnp);
16 #endif
17 }
```

Figure 4. Two Not-yet-discovered Bugs in Busybox

```
int main(int argc, char** argv){
 int r = rand() % 2;
 char *p;
 char c;
 scanf("%c", &c);
 switch (c){
  case 'W':
   if (r)
    p = *argv++, --argc;
   break;
 #ifdef CONFIG_FEATURE_HDPARM_HDIO_UNREGISTER_HWIF
  case 'U':
   if(!p)
     goto expected_hwif_error; //ERROR
   break;
 #endif /*CONFIG_FEATURE_HDPARM_HDIO_UNREGISTER_HWIF*/
 #ifdef CONFIG_FEATURE_HDPARM_HDIO_SCAN_HWIF
  case 'R':
   if(!p)
    goto expected_hwif_error;
 expected_hwif_error:
  printf("expected hwif value");

 #endif /* CONFIG_FEATURE_HDPARM_HDIO_SCAN_HWIF */
 }
 return 0;
}
```

Figure 5. A Run-time Feature-Interaction Bug in Busybox

E. Effectiveness in Detecting Bugs in the Wild (RQ3)

To evaluate the effectiveness of CoPro on the real-world, open-source projects, we ran it on the configurable systems *libpng* v0.89 and *xterm* v2.24 to detect interaction bugs. Interestingly, with CoPro, we were able to detect **4 interaction bugs that have not been reported/discovered before**. They have the same nature of *using variables/functions without declarations*. Let us discuss two of them in details. The other one can be found on our website [1].

In Figure 6, the code contains 2 bugs. The first one is observed when the option `PNG_READ_INTERLACING_SUPPORTED` or `PNG_WRITE_INTERLACING_SUPPORTED` is enabled (line 4) and `PNG_INTERNAL` is disabled (line 1). In this case, the function `png_set_interlace_handling` is declared (line 5), and `PNG_INTERLACE` (line 6) is used inside this function. Meanwhile, the constant `PNG_INTERLACE` (line 2) is declared only if `PNG_INTERNAL` is enabled. Thus, if `PNG_INTERNAL` is disabled, and either `PNG_READ_INTERLACING_SUPPORTED` or `PNG_WRITE_INTERLACING_SUPPORTED` is enabled, we will have a compiling error at line 6. The second bug occurs when both `PNG_READ_INTERLACING_SUPPORTED` and `PNG_WRITE_INTERLACING_SUPPORTED` are `F`. In this case, `png_read_image` use an undeclared function (line 10).

F. Time Complexity (RQ4)

We run our experiments on a computer with Intel Core i5 2.7GHz processor, 8GB RAM. The running time to analyze the most complex case that contains 43KLOC and 194 configuration options and rank 156 configurations is 211,020ms.

```
1  #if defined(PNG_INTERNAL)
2  #define PNG_INTERLACE  0x0002
3  #endif /* PNG_INTERNAL */
4  #if defined(PNG_READ_INTERLACING_SUPPORTED) ||
       defined(PNG_WRITE_INTERLACING_SUPPORTED)
5    int png_set_interlace_handling(png_structp png_ptr){
6      png_ptr->transformations |= PNG_INTERLACE;
7  }
8  #endif
9  void png_read_image(png_structp png_ptr){
10   int pass = png_set_interlace_handling(png_ptr);
11 }
```

Figure 6. Two Not-yet-discovered Bugs in libpng

G. Limitations and Potential Solutions

For the cases that CoPro did not rank well the buggy configurations, we found that the majority of them are not in the kinds of interaction-related defects listed in Section III. For example, a variable `x` is assigned a value `v` if option `A` is enabled, otherwise `x=v'`. Then, `x` is referred to in a feature controlled by option `B`. In this case, CoPro detects the interactions between those features. However, as a static technique, CoPro could not conclude which option selections are more suspicious. To overcome such limitation, one could use a dynamic analysis approach for configurable code [45].

Figure 7 shows a simplified bug in HTTPD (commit 9327311d30f) that CoPro did not rank well the buggy configurations. In Figure 7, a *use without assignment* is exposed when `APU_HAS_LDAP` and `APU_HAS_SHARED_MEMORY` are on. CoPro did not work since there is no feature where `rmm_lock` is assigned. Consequently, no *assign-use* interaction exists.

H. Extension to CoPro

Generally, to detect more kinds of bug such as in the above example, one can extend our set of conditions with the corresponding violations in Table III. One can define a new condition to detect this bug as follows: i) $\alpha(\texttt{APU\_HAS\_LDAP,T}) \cap \gamma(\texttt{APU\_HAS\_SHARED\_MEMORY,T}) \neq \emptyset = \{\texttt{util\_ldap\_cache\_init.rmm\_lock}\}$ and ii) there is no definition of `rmm_lock` in its scope, which is the function `util_ldap_cache_init`.

Interestingly, note that for this buggy system, CoPro ranked the configuration to reveal another flaw of *unused variable* (`rmm_lock`) when `APU_HAS_LDAP=T` and `APU_HAS_SHARED_MEMORY=F`.

In 14 cases out of 368 cases, the interactions that cause the interaction bugs are really detected, but the configurations that reveal the bugs are still ranked lower than others. The reason for these cases is that other configurations containing more suspicious selections that actually do not cause the bugs. To faster detect the bug in these situations, one can apply the *Additional Priortization* strategy [22] to rank the set of configurations according to their numbers of potential bugs in an incremental manner. By this strategy, the next configuration to be selected is the one containing the largest number of

```
void apr_rmm_init(char* rmm_lock){
 printf("%s\n", rmm_lock);
}
#ifdef APU_HAS_LDAP
void util_ldap_cache_init(){
  char* rmm_lock;
#ifdef APR_HAS_SHARED_MEMORY
  apr_rmm_init(rmm_lock); // ERROR: rmm_lock
      uninitialized
#endif
}
#endif
```

Figure 7. CoPro did not rank well buggy configurations

potential bugs that have not been contained by the previous selected configurations in the previous steps. Moreover, for the interaction bugs relevant to pointers and external data files, one can define new rules to add to our framework.

VI. Related Work

CoPro is most closely related to the work by Al-Hajjaji *et al.* [3] on **similarity-based Prioritization** (SP). The key idea of SP approach dissimilar test sets are likely to detect more defects than similar ones [3]. In SP, the configuration with the maximum number of features is selected to be the first one under test and the next configuration must have the minimum number of similar features as the previously selected configuration. In comparison, SP does not analyze the nature of feature interactions, while CoPro does. This avoids the problem in SP that the different features to be considered next might not be the ones that potentially causes violations.

CoPro is also related to the work on **configuration selection** approaches to reduce the number of configurations to be tested [42]. They focus on the step before configuration prioritization, therefore the resulting set of configurations is not ranked as in CoPro. The t-wise (i.e., k-way) sampling algorithm covers all combinations of t options [28], [35], [46], [48], while pair-wise checks all pairs of configuration options [40], [47]. Recent study by Medeiros *et al.* [42] showed that realistic constraints among options, global analysis, header files, and build-system information influence the performance of most sampling algorithms substantially; and several algorithms are no longer feasible in practice. Importantly, they lack configuration prioritization, thus, developers need to spend efforts to perform QA on all the variants.

CoPro is also related to **Variability-aware (VA) analysis** [37]. VA analysis is a variation of a traditional analysis that considers the variability in the configurable code. The variability-aware analysis techniques have been proposed for type checking [16], [31], [37], [54], model checking [19], [34], data-flow analysis [11], [12], [37], and other analyses [21] on multiple compile-time configurations of a system at a time. The main drawback of this approach is that it cannot reuse existing static analysis tools, and each type of analysis must be rewritten in a variability-aware fashion. For example, to detect `NULL` exception, one must rewrite such an analysis to consider all different configurations in a configurable code. In our experiment, we connect CoPro with an existing bug detection tool to work on configurable code. Variability-aware execution [45], [43] explores multiple paths of execution at the same time to detect feature-interaction bugs. However, it suffers scalability issue.

Several approaches were proposed to detect **feature interactions** [6], [25]. Verification [7] is also used to detect feature-interaction bugs. Other prioritization approaches aim for *statement coverage* [52], [53] via static checkers. The issue is that computing an optimal solution for the coverage problem is NP-hard, and including each block of optional code at least once does not guarantee that all possible combinations of individual blocks of optional code are considered [42]. To avoid finding optimal coverage solution, the *most-enabled-disabled* [52] algorithm checks two samples independently of the number of configuration options. When there are no constraints among configuration options, it enables all options and then it disables all configuration options. One-(enabled/disabled) algorithm [2] enables/disables one configuration option at a time. Despite different levels of heuristics, they do not analyze the entities in source code.

Several pproaches are aimed for **testing for configurable systems** [14], [20], [26], [39]. In product-line testing [49] and framework testing [15] it is a common strategy to unit test components or plug-ins in isolation, while integration tests are often neglected or performed only for specific configurations. Greiler *et al.* suggest shipping test cases with plug-ins and running them in client systems [26]. In essence, this postpones tests of configurations until the configuration is actually used.

Other approaches have been proposed for static analysis of product lines [11], [12], [16], [19], [21], [31], [54], [55]. Researchers explore to represent and reason about partial but finite configuration spaces compactly with BDDs or SAT solvers (as used in our variability contexts) [8], [31], [44], choices of structures [23] and complex structures [24], [37].

VII. Conclusion

We propose CoPro, a novel formulation of feature-interaction bugs via common program entities enabled/disabled by the features. Leveraging from that, we develop efficient feature-interaction-aware configuration prioritization technique for a configurable system by ranking the configurations according to their total number of potential bugs. We evaluated CoPro in two complementary settings: detecting configuration-related bugs in a benchmark and a real-world open-source systems. CoPro outperforms the other techniques in which in 78% of the cases, it ranks the buggy configurations at the top 3 positions. Interestingly, it is able to detect 17 not-yet-discovered, high-degree, feature-interaction bugs.

Acknowledgment

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

REFERENCES

[1] . https://doubledoubleblind.github.io/copro/.

[2] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 421–432, New York, NY, USA, 2014. ACM.

[3] Mustafa Al-Hajjaji, Thomas Thüm, Jens Meinicke, Malte Lochau, and Gunter Saake. Similarity-based prioritization in software product-line testing. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 197–206, New York, NY, USA, 2014. ACM.

[4] Apache Httpd. http://httpd.apache.org/.

[5] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *JOURNAL OF OBJECT TECHNOLOGY*, 8(5).

[6] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. Exploring feature interactions in the wild: The new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, FOSD '13, pages 1–8, New York, NY, USA, 2013. ACM.

[7] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of Feature Interactions Using Feature-aware Verification. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 372–375, Washington, DC, USA, 2011. IEEE Computer Society.

[8] Don Batory. Feature models, grammars, and propositional formulas. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20, Berlin/Heidelberg, 2005. Springer-Verlag.

[9] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, Dec 2013.

[10] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wkasowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

[11] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spllift: Statically analyzing software product lines in minutes instead of years. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 355–364, New York, 2013. ACM Press.

[12] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 13–24, New York, 2012. ACM Press.

[13] Busy Box. https://busybox.net/.

[14] Isis Cabral, Myra B. Cohen, and Gregg Rothermel. Improving the testing and testability of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, pages 241–255, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] Sheng Chen, Martin Erwig, and Eric Walkingshaw. Extending type inference to variational programs. Technical report (draft), School of EECS, Oregon State University, 2012.

[16] Sheng Chen, Martin Erwig, and Eric Walkingshaw. Extending type inference to variational programs. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 2013.

[17] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 321–330, New York, NY, USA, 2011. ACM.

[18] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 335–344, New York, NY, USA, 2010. ACM.

[19] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-Francois Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 335–344, New York, 2010. ACM Press.

[20] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 129–139, New York, NY, USA, 2007. ACM.

[21] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220, New York, 2006. ACM.

[22] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb 2002.

[23] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(1):6:1–6:27, 2011.

[24] Martin Erwig and Eric Walkingshaw. Variation programming with the choice calculus. In *Generative and Transformational Techniques in Software Engineering IV*, pages 55–100. Springer Berlin Heidelberg, 2013.

[25] Brady J. Garvin and Myra B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 90–99, Washington, DC, USA, 2011. IEEE Computer Society.

[26] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 244–254, Piscataway, NJ, USA, 2012. IEEE Press.

[27] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modeling and model checking software product lines. In *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS '08, pages 113–131, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] Martin Fagereng Johansen, Oystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 46–55, New York, NY, USA, 2012. ACM.

[29] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[30] Christian Kästner. Virtual separation of concerns: toward preprocessors 2.0. *it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 54(1):42–46, 2012.

[31] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(3):14:1–14:39, 2012.

[32] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824, New York, NY, USA, 2011. ACM.

[33] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 25–32, New York, NY, USA, 2010. ACM.

[34] Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 269–280, Los Alamitos, CA, 2009. IEEE Computer Society.

[35] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. Ipog-ipog-d: Efficient test generation for multi-way combinatorial testing. *Softw. Test. Verif. Reliab.*, 18(3):125–148, September 2008.

[36] libpng. http://www.libpng.org/.

[37] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 81–91, New York, NY, USA, 2013. ACM.

[38] Linux Kernel. https://www.kernel.org/.

[39] Ivan Do Carmo Machado, John D. Mcgregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. On strategies for testing software product lines: A systematic literature review. *Inf. Softw. Technol.*, 56(10):1183–1199, October 2014.

[40] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical pairwise testing for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 227–235, New York, NY, USA, 2013. ACM.

[41] Marlin. http://marlinfw.org/.

[42] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 643–654, New York, NY, USA, 2016. ACM.

[43] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 483–494, New York, NY, USA, 2016. ACM.

[44] Marcílio Mendonça, Andrzej Wkasowski, and Krzysztof Czarnecki. SAT-based analysis of feature models is easy. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 231–240, New York, 2009. ACM Press.

[45] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 907–918, New York, NY, USA, 2014. ACM.

[46] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, February 2011.

[47] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, pages 196–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[48] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 459–468, Washington, DC, USA, 2010. IEEE Computer Society.

[49] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin/Heidelberg, 2005.

[50] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 347–350, Washington, DC, USA, 2008. IEEE Computer Society.

[51] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct 2001.

[52] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static analysis of variability in system software: The 90,000# ifdefs issue.

[53] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration coverage in the analysis of large-scale system software. *SIGOPS Oper. Syst. Rev.*, 45(3):10–14, January 2012.

[54] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104, New York, 2007. ACM Press.

[55] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, April 2012.

[56] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.

[57] xterm. https://invisible-island.net/xterm/.

[58] Linbin Yu, Yu Lei, Raghu N Kacker, and D Richard Kuhn. Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 370–375. IEEE, 2013.

[59] Pamela Zave. Programming methodology. chapter An Experiment in Feature Engineering, pages 353–377. Springer-Verlag New York, Inc., New York, NY, USA, 2003.