Tracking Load-time Configuration Options

Max Lillack, Christian Kästner, and Eric Bodden

Abstract—Many software systems are highly configurable, despite the fact that configuration options and their interactions make those systems significantly harder to understand and maintain. In this work, we consider load-time configuration options, such as parameters from the command-line or from configuration files. They are particularly hard to reason about: tracking configuration options from the point at which they are loaded to the point at which they influence control-flow decisions is tedious and error-prone, if done manually. We design and implement LOTRACK, an extended static taint analysis to track configuration options automatically. LOTRACK derives a configuration map that explains for each code fragment under which configurations it may be executed. An evaluation on Android apps and Java applications from different domains shows that LOTRACK yields high accuracy with reasonable performance. We use LOTRACK to empirically characterize how much of the implementation of Android apps depends on the platform's configuration options or interactions of these options.

Index Terms—Variability mining, Configuration options, Static analysis

1 INTRODUCTION

SOFTWARE has become increasingly configurable to support different requirements for a wide range of customers and market segments [56]. Configuration options can be used to support alternative hardware, cater for backward compatibility, enable extra functionality, add debugging facilities, and much more. While configuration mechanisms allow end users to use the software in more contexts, they also raise the software's complexity for developers, adding more functionality that needs to be tested and maintained. Even worse, configuration options may interact in unanticipated ways and subtle behavior may hide in specific combinations of options that are difficult to discover and understand in the exponentially growing configuration space. Configuration options raise challenges since they vary and thus complicate the software's control and data flow. As a result, developers need to trace configuration options through the software to identify which code fragments are affected by an option and where and how options may interact. Overall, making changes becomes harder because developers need to understand a larger context and may need to retest many configurations.

There are many strategies to implement configuration options, but one common way is to use *load-time parameters* (command-line options, configuration files, registry entries, and so forth): Parameters are loaded and used as ordinary values within the program at runtime, and configuration decisions are made through ordinary control statements (such as if-statements) within a common implementation. Load-time parameters may be problematic regarding traceability between the point of accessing configuration options and their effect on the code. Beyond very simple uses of load-time configuration options, identifying the code fragments implementing an option requires tedious manual effort and, as our evaluation confirms, is challenging to get right even in medium-size software systems.

In this work, we propose LOTRACK,[1] a tool to statically track configuration options from the place where they are loaded in the program to the code that is directly or indirectly affected. Specifically, LOTRACK aims at *identifying all code that is included if and only if a specific configuration option or combination of configuration options is selected*. The recovered traceability can support developers in many maintenance tasks [20], but, in the long run, can also be used as input for further automated tasks, such as removing a configuration option and its use from the program, translating load-time into compile-time options, or guaranteeing the absence of interactions among configuration options. In contrast to slicing [55], especially forward slicing, which determines *whether* a statement's execution depends on a given value, LOTRACK determines *under which* configurations, i.e., a set of selected configuration options, a given statement is executed.

To track configuration options precisely, we exploit the nature of how configuration options are typically implemented. Although a forward-slicing algorithm can identify all code potentially affected by a configuration option, directly or indirectly, in practice, it will frequently return slices that are largely overapproximated, due to hard-to-handle programming features such as aliasing, loops and recursion. To increase precision, we exploit the insight that *configuration options are typically used differently from other values in the code:* Values for configuration options are often passed along unmodified and are used in simple conditions, making their tracking comparatively easy and precise. Finally, usually only few configuration options are used in any given part of a program. Technically, LOTRACK extends a context, flow, object and field-sensitive static *taint analysis* [4] to build a *configuration map* describing how code fragments depend on configuration options.

This paper evaluates LOTRACK in the context of Android apps and Java applications. Android apps are interesting

- *M. Lillack is with University of Leipzig, Germany.*
- *C. Kästner is with the School of Computer Science at Carnegie Mellon University, USA.*
- *Eric Bodden is with Heinz Nixdorf Institute, Paderborn University & Fraunhofer IEM, Germany.*

1. https://github.com/MaxLillack/Lotrack

subjects for studying configuration options, because the Android platform has a reputation for being diverse and fragmented with many different platform versions and hardware features [35]. Android apps query a fixed set of configuration options given by the framework to dynamically switch between implementations or disable functionality if the corresponding feature (e.g., Bluetooth support) is not available on a device. We also evaluate LOTRACK on Java applications to show the generality of the approach and tool. Unlike Android apps which can use a set of options defined by the framework, Java applications access load-time configuration options in many different ways and for a wide range of use cases. Especially command-line Java applications are interesting examples for configuration maps because they use options to conditionally enable/disable certain features of a program. We apply LOTRACK to different types of applications as well as applications from different domains.

In a corpus of Android apps and Java applications, we track how configuration options are used and how much code is devoted to implement optional functionality. We find that most Android apps use standard configuration options given by the framework to optionally include code. We estimate an average of 1% of the apps' source is executed depending on configuration options. For Java applications we find that the use of configuration options is very diverse: some applications use options only locally, whereas in other applications options affect a significant portion of the code.

Even if the share of configuration-dependent code is small, creating a configuration map manually is often not a viable option. Tracking configuration options manually is a tedious, error-prone, and time-consuming task. As we will show in our evaluation, configuration dependent code is scattered throughout the program and developers often fail to identify all configuration-dependent code correctly.

In summary, this paper presents the following contributions:

1) an encoding of the problem of tracking configuration options as a taint-analysis problem,
2) a description of how to make use of common characteristics of configuration values in programs to increase the precision of the analysis,
3) an implementation based on FLOWDROID [4], able to handle Java/Android source code and bytecode,
4) an empirical evaluation demonstrating the precision and recall of our implementation as well as an overview of configuration option usage based on a sample of 100 open-source Android apps,
5) the extension of the approach and corresponding implementation in LOTRACK to support integer-based configuration options,
6) a demonstration of LOTRACK on Java applications from different domains, and
7) an empirical evaluation comparing the approach to traditional approaches of static program slicing.

This article extends our previous approach to tracking load-time configuration options [28]. Compared to our previous work, we add contributions 5-7, provide a revised and more detailed presentation of the approach, and a significant revision and extension of the tool regarding scalability. The new contributions show that the approach is applicable to a wide range of applications and types of configuration options beyond the Android apps and Boolean options used in the previous study. The added support for integer options (now backed by an SMT solver) increases the coverage of the precise analysis; other types of configuration option, e.g. strings, are still tracked in an imprecise way. Our experiments show that a configuration map provides a more focused and detailed result than a comparable program slice.

2 PROBLEM STATEMENT

Our goal is to trace configuration options to the code fragments implementing them. That is, we want to find all code that is executed if and only if specific configuration values are set. For example, in an Android app, we might want to find all source code bound to the availability of Bluetooth or to functionality only active on devices running Android 4.4 or higher.

Technically, we seek to establish a *configuration map*, which maps every code fragment to a *configuration constraint* describing for which configurations the code fragment can be executed, that is, which configuration options or combinations of options need to be selected or deselected. We describe the configuration constraint as a formula over configuration decisions. A configuration constraint is a selection for a specific configuration option, such as $\mathit{Bluetooth} = \mathit{on}$ (abbreviated to $\mathit{Bluetooth}_+$ for Boolean options) or $\mathit{SDKVersion} \geq 4.4$. If we only know that a configuration option O is involved, but we are unable to figure out more precisely how, we write $O_?$ as configuration constraint. A configuration constraint may describe many configurations; for example, $\mathit{Bluetooth}_+ \wedge (\mathit{SDKVersion} \geq 4.4)$ describes the set of configurations in which Bluetooth is enabled and a newer SDK version is used. In Fig. 1, we illustrate a configuration map for a simple excerpt from the Adblock Plus[2] app in which the configuration constraint for each statement is written to the left of the line. While this is a simple excerpt for demonstration purposes, the whole app uses the shown field six times and in four different classes. Such a use of a configuration-related field shows the scattered nature of the configuration option's implementation.

A possible simpler variant of the configuration map, which might be simpler to compute, is a mapping from statements to a set of configuration options on which their execution depends. However, compared to this simple variant, the added precision shows *how* options influence whether a statement is executed and there are examples where this information is necessary:

- A dependency on the option *SDK version* itself is not necessarily important to a developer, only a more precise constraint *SDK<9* indicates code specific for older versions which can be removed when the app no longer supports this version.
- Program comprehension is supported because precise results show how exactly an option is influencing the program.
- Precise results are needed for test case generation. Similar to the example above, a constraint *SDK < 9* can be used to construct a test suite with *SDK=8* and *SDK=9*.

2. https://github.com/adblockplus/adblockplusandroid

```
01 class ProxyService {
02   static boolean NATIVE_PROXY_SUPPORTED = Build.VERSION.SDK_INT >= 12;
03   public void onSharedPreferenceChanged() {
04     String ketHost;
05     if (!NATIVE_PROXY_SUPPORTED) {
SDK<12 06       ketHost = getString(R.string.pref_proxyhost);
       07       ...
08     }
09     String command = path + " -host ";
10     String result = RootTools.sendShell(command + ketHost);
11     ...
12   }
13 }
14 class ConfigurationActivity {
15   public void onHelp(View view) {
16     Intent intent;
17     if (ProxyService.NATIVE_PROXY_SUPPORTED)
SDK>=12 18       intent = new Intent(this, ProxyConf...);
19     else
SDK<12 20       intent = new Intent(Intent.ACTION_VIEW, uri);
21     startActivity(intent);
22   }
23 }
24
```

Fig. 1: Example from Adblock Plus app and expected configuration map.

A configuration map can support developers in performing maintenance tasks or in reasoning about the implementation. Developers can look up all code fragments implementing a specific configuration option and can investigate how two configuration options relate. For instance, in prior work, we and others have shown how background colors and views/projections highlighting options can significantly improve developer productivity, especially if the implementation of configuration options is scattered throughout multiple locations [5], [14], [26]. A configuration map simplifies otherwise potentially daunting tasks, such as removing an obsolete option from the code [8], refactoring the scattered implementation of an option into a module [1], [22], [29], changing the binding time of a configuration option between compile-time and load-time [43], or determining test-adequacy criteria with configuration coverage [50]. With a precise configuration map, one could even determine that two configuration options can never interact and thus could establish that one does not need to test their interactions. For the example shown in Fig. 1, the configuration map highlights the scattered implementation fragments implementing the option's functionality and supports quick navigation. Note that, in contrast to slicing [55], our configuration map does not include statements that use configuration values or values influenced by them (e.g., Line 21 in our example), but only code blocks included or excluded by configuration-related control-flow decisions.

There are many different strategies to implement configuration options [3], some of which allow us to extract a configuration map easily. For example, when providing optional functionality as plug-ins to frameworks such as Eclipse and Wordpress, one can locate the corresponding implementation in those plug-ins. Similarly, using conditional compilation, for example using the C preprocessor's #ifdef directives, despite all criticism [13], [48], enables a simple static localization of all scattered code fragments implementing an option with a simple search over those directives [3], [8], [47]. [3] Unfortunately, for load-time configuration options there is no such simple static extraction, because configuration happens after compile time and because a simple syntactic analysis is insufficient to distinguish configuration values from other runtime values.

In this work, we thus design a static analysis that approximates a *configuration map for load-time configuration options* by tracking each configuration option from the point at which it is loaded to the control-flow decisions that include or exclude a code fragment depending on the option's value.

To scope our approach, we make the following assumptions:

- Configuration options are set at program load time and do not change during the execution of the program, hence reading the same configuration value multiple times will always yield the same result. Yet, the read configuration value may be assigned to variables and those variables' values may change during runtime.
- The API calls to load configuration values are known and can be identified syntactically (e.g., the read from field `SDK_INT` in Fig. 1). How these options are identified is outside the scope of this paper. Possible strategies include manual identification by reading source code and documentation and using existing heuristics and static analysis tools [38].
- After being read from the API, configuration values may be assigned to variables or fields and may be propagated or processed in arbitrary ways in the program. Configuration options may trigger data dependencies in other variables and only indirectly influence control-flow decisions. This is a relaxation compared to approaches for preprocessor-based configuration which

3. Compile-time configuration mechanisms can still trigger runtime decisions, for example by using a macro with alternative compile-time values to initialize a variable that is subsequently used in runtime control-flow decisions. Current techniques do not discover these dependencies crossing binding times; more advanced static analyses would be required, similar to what we propose for load-time configurations in this paper.

only detect direct dependencies between configuration option and code.

By tracking configuration options in a program, we are essentially tracking all control and data dependencies of a value through arbitrary computations. Since such static computation is undecidable (Rice's theorem [42] states that any non-trivial property about a program is undecidable), our approach relies on standard static-analysis techniques, conservatively abstracting over concrete values, similar to, e.g., program slicing [55]. In general, one might think that too coarse abstractions could easily yield useless overapproximations, where essentially every code fragment is potentially influenced by every configuration option. As we observed in practice, however, in many programs configuration options are used in limited ways. In particular, one can tailor static program analyses because configuration options often exhibit the following common characteristics:

- Configuration options often have a small finite domain, in many cases they have just two possible values, which makes it feasible to track concrete values and efficiently reason about expressions over configuration values.
- Configuration options are commonly reassigned and propagated throughout the program, but they are rarely changed once they are loaded.
- Configuration options often occur in control-flow decisions (e.g., if-statements), but they rarely are involved in more complex computations. For example, one might compute the square root of a regular input, but rarely of a configuration option.

The context, flow, object, and field-sensitive taint analysis underlying LOTRACK enabled us to track the use of configuration options.

3 APPROACH

The idea of LOTRACK is to use a taint analysis to track configuration options through the code and identify when control-flow decisions depend on tainted values. A taint analysis is a data-flow analysis typically used in security research. For example, to detect information leaks, a private value is marked as tainted and all values derived from this value (directly or indirectly) are tainted as well, allowing one to recognize when tainted private values are used in contexts where they should not (e.g., sent over a network). LOTRACK uses a taint analysis in a slightly different way: It taints all values resulting from reading a configuration option or from a computation with a tainted value; when a tainted value occurs in a control-flow decision, one knows that all code in this branch may depend on this configuration option. To reduce overapproximation and produce an accurate configuration map, for select configuration options LOTRACK additionally tracks specific values as conditional taints. This extension of the taint analysis, which we call *value tracking*, is feasible for our problem because configuration values are mostly used unchanged or within simple operations, e.g. comparisons to constants. In Section 3.1, we show that we can create a configuration map via taint analysis. We extend this approach with value tracking, which we explain in detail in Section 3.2 and then formalize the approach in Section 3.3.

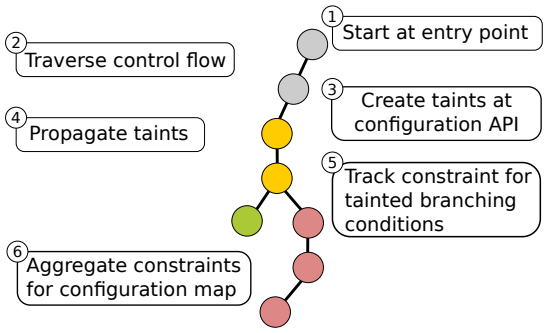


Fig. 2: Visualization for the configuration tracking process.

3.1 Taint Analysis for Configuration Options

To outline the basics of our approach, we show how we use static taint analysis for tracking configuration options and how it differs from a typical taint analysis. First, we will explain the two important aspects of taint analysis, taint creation and propagation. Second, we will describe how, for each taint and statement, we also maintain a *constraint* as part of the taint information.

To better illustrate the steps in the approach, we show a high-level overview in Fig. 2.

3.1.1 Taint creation

The taint analysis will create new taints at statements accessing a configuration option and at assignments in which a tainted variable is read. A taint will contain the information with which configuration option the variable is associated. The analysis creates an *implicit flow* when a tainted value is used in a branching statement. Such an implicit flow indicates that the reachability of the current statement is dependent on a configuration option. Such a taint is not linked to a variable but still has the link to a configuration option. The implicit flow taint is only valid within the branches of the branching statement. More importantly, an implicit flow can induce the creation of additional taints: At assignment statements where an implicit flow is present the assigned variable will be tainted to capture an indirect dependency on a configuration option. The configuration constraint for the configuration map is derived from implicit flow taints representing the reachability of statements in the program.

As a running example, we will use the small piece of code shown in Fig. 3, to the left and the right of the code the figure shows two configuration maps with different levels of precision. We will first explain how to create the simple variant of the configuration map shown on the left; in Section 3.2, we will extend our analysis with value tracking for the more detailed version of the configuration map shown on the right side. At Line 1, LOTRACK will check its list of methods and fields that are used to read configuration options. In this case the field `Options.SDKVersion` is associated with the option *SDK*. Therefore, LOTRACK taints variable `v` with the option *SDK*. The created taint represents any valid configuration value $(8, 9, 10, ...)$ of the option *SDK*. At the if-statement in Line 4 of Fig. 3, an implicit flow is created because the branching condition depends on a tainted value. At Line 5, this implicit flow then induces the

```
                01 int v = Options.SDKVersion;
                02 boolean wifiOn = Options.WIFI_ON
                03 boolean proxySupported;
                04 if(v > 8)
  {SDK}         05   proxySupported = true;              SDK>8
                06 else
  {SDK}         07   proxySupported = false;             SDK≤8
                08 if(proxySupported && wifiOn)
{SDK,WIFI}      09   [...]                          SDK>8 ∧ WIFI
```

Fig. 3: Example for access and use of configuration options. On the left side, the configuration map *without* value tracking is shown, on the right side the configuration map *with* value tracking is shown.

creation of a taint for variable `proxySupported`, indicating that the value of this variable may also depend on option *SDK*. A similar taint is created at Line 7. The configuration map can be derived directly from implicit flow taints: The reachability of the statements at Line 9 may depend on the options *SDK* and *WIFI*, even though there is no data flow from the API to the variable `proxySupported`.

3.1.2 Taint propagation

The taint analysis propagates taints inter-procedurally along control-flow edges to all values that directly or indirectly depend on this value, considering both control-flow and data-flow dependencies.

Statements accessing a configuration option correspond to the *sources* of conventional taint analysis. There is no equivalent to a *sink*, where the taint propagation would stop, instead, every taint is propagated as far as possible.

A common problem with taint analysis is how to handle native functions and environment interactions. For a sound analysis, unless one knows how information flows through the environment, one has to assume the worst, i.e., that every value read from the environment may be tainted (i.e., depends on some configuration option), often leading to massively overapproximated results. For results of native-method calls or environment interactions, we allow false negatives and only create taints if those calls or interactions have been parameterized with a tainted value. This simplification is grounded in the assumption that configuration options are mostly used in simple ways so that false negatives should be rare. In fact, handling of native functions and environment interactions are customizable to different levels of strictness, and the underlying FLOWDROID tool [4] supports such customizations through its configuration.

3.1.3 Constraint Calculation

The taint analysis tracks whether a variable's value is based on a configuration option, but it does not consider variability in the program due to configurations. For this, LOTRACK additionally tracks a constraint as part of each taint. This constraint describes the set of configurations for which the corresponding taint is valid. Since this difference is not obvious, we illustrate the need for both kinds of information using a minimal example: `if(OptionA) {b=OptionB}`. Here, a taint will contain the information that variable `b` is related to OptionB. Additionally, we use the constraint to track that this taint is only active if OptionA is enabled.

To create a configuration map, e.g., as shown on the left side of Fig. 3, LOTRACK creates taints for all configuration options and maps each code fragment whose execution is dependent on a tainted variable to the configuration options associated with the taint. Intuitively, every time a tainted value associated with some option occurs in the expression of an if-statement (or other control-flow decision), all statements in the *then* and *else* branches depend on the configuration option and thus are associated with it. We can associate complete methods or classes with configuration options if every statement they contain are exclusively used in paths guarded by tainted conditionals.

3.2 Extending Taint Analysis for Tracking Configuration Values

The simple taint-based analysis above creates a map between code fragments and all involved configuration options. However, it does not tell *how* configuration options influence the selection of a code fragment. In our example (Fig. 3), we would ideally like a more precise configuration constraint to know that Line 5 is only executed if $SDK > 8$ instead of only knowing that it *somehow* depends on *SDK*. To that end, we extend the taint analysis to track configuration *values* instead of only configuration options. While such analysis can be very expensive in general, the way configuration options are used allows us to scale such more precise analysis to many programs.

We first explain the individual parts of value tracking and illustrate them using specific examples while formalizing the approach in Section 3.3.

3.2.1 Extension to Taint Analysis

Value tracking is an extension to taint analysis to extract precise constraints of the configuration map. For this, LOTRACK maintains a precise constraint under which configuration a taint is propagated. Also, LOTRACK does not propagate all taints directly, but analyzes, restricts, and merges constraints at control-flow decisions. To ensure a correct and fast execution, any operation on the constraints is reduced to a logic formula that is solved by a standard SMT solver.

With value tracking, we track more specific information about which variable can have which (concrete or symbolic) values through three different kinds of taints:

1) When possible, we track a concrete value for select variables (e.g., true, false, 1, 2) using a *value taint*.

2) We track a symbolic value representing a configuration decision (e.g., whether GPS is enabled) using an *option taint*. For configuration options with small finite domains, we could track all possible configuration values separately with value taints. We introduce option taints to efficiently handle configuration options with larger domains.
3) We track an unknown (symbolic) value of a variable with an *imprecise taint* if the value is somehow affected by a configuration option. For this unknown value, we note the single configuration option to which it is related. If a variable's unknown value may be related to different options, we use multiple taints for each option. Imprecise taints are equivalent to our initial taint analysis without value tracking (Section 3.1) and we fall back on them when value tracking is intractable. Note that we still do not taint values that are computed independent of configuration decisions. In Section 3.2.5 we provide examples to demonstrate how imprecise taints are used.

3.2.2 Creation of Constraints

LOTRACK creates a *constraint* when a tainted value is used in the condition of an if-statement, the constraint is used for implicit taints and subsequently propagated to other taints. A constraint describes for which configurations a taint is defined. To create precise constraints, LOTRACK analyzes the *if* condition and the available taints for variables used in the condition. If a condition contains a tainted variable, the condition is statically evaluated with respect to the information on configuration options in the taints. The variable in the condition is replaced with the value from available taints. The resulting expression, in conjunction with the taint's constraint, is passed to the solver to determine its satisfiability. If there are multiple taints for the variable, the resulting constraint will be the disjunction of the expression created from each taint. To determine the constraint of the fall-through branch LOTRACK negates the constraint.

To illustrate this creation of constraints with value tracking, we show taints created as part of our running example in Fig. 4. Here, we have an *option taint* used in a condition (Line 3) for variable `v` and option *SDK*. From the condition (`v > 8`), a constraint ($SDK > 8$) is created, essentially taking the condition from the code and replacing the variable with the taint value. For the fall-through edge, the constraint is negated resulting in $\neg(SDK > 8)$. Since the if-statement uses a tainted variable, the implicit flow induces the creation of taints for assignments in the two branches (Lines 4 and 6). At Line 7, there are two taints (with values true and false) present for the variable `proxySupported` used in the condition. The Boolean condition is only satisfiable for the taint with value *true*, resulting in constraint $SDK > 8$ for Line 8.

3.2.3 Propagation of Constraints

Tracking constraints is an extension to the underlying taint analysis. Constraints are propagated as parts of taints. Constraints change depending on the control-flow of the program and the presence of other taints.

If a control-flow decision depends on a tainted value, we derive constraints for the control-flow branches by evaluating the condition, as described before. Since the branches of the control-flow decision will only be executed if the condition holds, we need to *restrict* the constraints of taints propagated along such an edge. The new constraint is the conjunction of the taint's previous constraint and the control-flow branch's constraint. Fig. 5 illustrates the conjunction of constraints for nested if-statements.

If several taints with the same combination of variable and taint value reach the same statement, for instance, at a control-flow merge point, the constraints for these taints are combined as disjunctions, leading to a *less restrictive* constraint. This rule ensures that a constraint represents all possible paths which can lead to the existence of the taints at the current statement. For example, two taints for the same variable and the same value but with different constraints *DEBUG* and $\neg$*DEBUG* are merged into a single taint with the constraint *true*.

The propagation terminates when there are no more new taints to propagate, we discuss the specifics of the termination in Section 3.3.4.

3.2.4 Building the Configuration Map

At the fixed point, the analysis has gathered taints with constraints for each reachable statement in the program. To create the configuration map, LOTRACK creates a single configuration constraint for each statement: the constraints of all taints at a statement are combined by disjunction. This configuration constraint represents the weakest constraint that must hold so that at least one taint is present at this statement. For example, in Fig. 4 the taints shown will be propagated to Line 7. To build the constraint for this line, LOTRACK creates the disjunction of the constraints of all the taints $true \vee (SDK > 8) \vee \neg(SDK > 8)$ which results in *true*.

3.2.5 Imprecise constraints

With value tracking, one can directly model constraints that are supported by the underlying solver. With the SMT solver used by LOTRACK one can support options with Boolean and integer domains. For a constraint with an unknown value of configuration option O we use the notation $O_?$ to indicate an *imprecise* relation to the option O that cannot be expressed more precisely. For example, $Version_?$ indicates an unknown relation to configuration option *Version*. We will now explain in detail how imprecise configuration options are used in LOTRACK.

The motivation behind imprecise constraints is that our analysis should be as precise as possible but for cases where one cannot statically reason about some properties of the program, we want to fall back on a solution where we *know* that we are losing precision and still use as much information as possible. First, imprecise constraints are used for configuration options with an unsupported domain, e.g., String options. Second, imprecise constraints are used as a fallback mechanism when a variable can no longer be tracked precisely. For example, if a taint flows through the environment and the result influences the configuration map, this influence is modeled using an imprecise constraint. This fallback is important for cases where our assumptions about how configuration options are used (Section 2) do not hold.

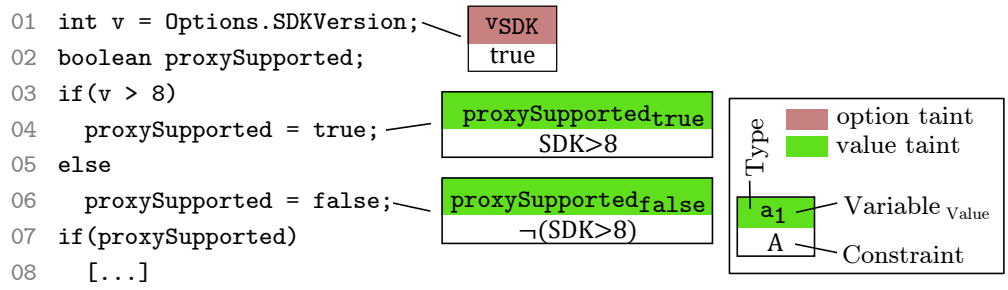
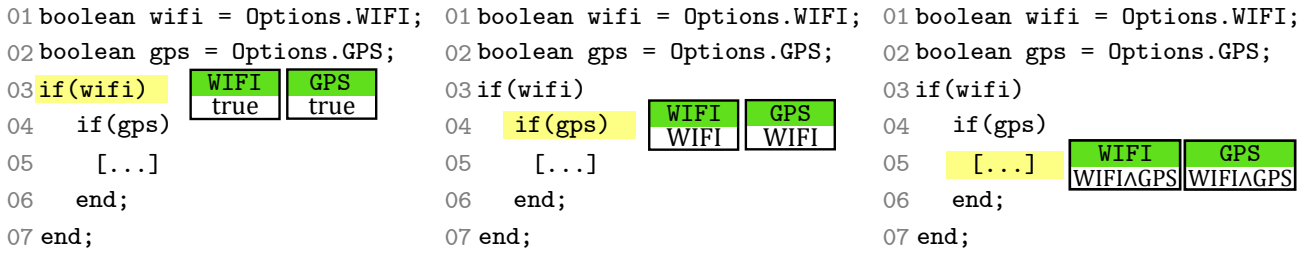


Fig. 4: Example based on Fig. 3 annotated with important taints.



(a) Taint for variables `wifi` and `gps` with initial constraints true

(b) At the first if-statement (Line 3) constraint WIFI is created.

(c) Conjunction of second *if*'s constraints (GPS) and existing constraint (WIFI)

Fig. 5: Conjunction of Constraints.

Instead of using a generic *unknown* value, each symbol in an imprecise constraint is still associated with a configuration option. This follows our idea of using available information even in cases where one cannot produce precise constraints. An imprecise constraint is a logical formula which contains many symbols and thus represents an imprecise relation to more than one option. Imprecise symbols are used as an abstraction over all kinds of sources of uncertainty (unsupported option domain, data flow through environment). Therefore, we cannot distinguish how strong the association of an imprecise symbol to its configuration option is; it can range from a strong equivalence even to a non-existing relation.

To represent their respective constraints LOTRACK creates a new and unique imprecise symbol for every condition in the control-flow. In our notation, we use subscripts $\alpha, \beta, \gamma, ...$ to distinguish different symbols for the same configuration option. Unique symbols allow LOTRACK to join the resulting constraints when taints with contradicting constraints are merged, e.g., $A_\alpha \vee \neg A_\alpha = true$.

The symbols used in imprecise constraints represent a condition as a whole and are never simplified with respect to the original condition. This allows us to model any condition whether the used operation is supported by the used solver or not. For example, $a \leq unknown$ and $a < unknown$ are represented as A_α and A_β. Then, a constraint $A_\alpha \wedge A_\beta$ is *not* simplified to A_α, although this would be possible given the semantics of the two conditions and assuming the same value of *unknown* at both locations. A similar case for precise constraints, based on the conditions `a <= 1` and `a < 2`, would be simplified to `a <= 1` to improve the readability of the resulting configuration map.

Conceptually, our approach uses a three-valued logic (true, false, unknown), but it is implemented using Boolean semantics with additional Boolean symbols to represent unknown values. The advantage of this approach is that we can use a standard solver, though it can lead to large constraints, which increases the solver's runtime.

```
                01 String url = Options.HostURL;
                02 if(url.startWith("https"))
HOST_α          03   [...]
                04 else
¬HOST_α         05   [...]
                06 int v = Options.SDKVersion
                07 int a = Math.max(v, random());
                08 if(a == 8)
SDK_α           09   [...]
```

Fig. 6: Imprecise tracking of configuration options.

For a user, a configuration map with imprecise constraints can still provide information about the effect of configuration options even if the tool is unable to provide more precise results. Additionally, imprecise constraints indicate how a program may be restructured to support a more precise analysis of configuration options. For example, some programs use string options where an integer or enum type would be more appropriate, such a refactoring could improve the readability of the code, and precision of the configuration map.

To illustrate the handling of imprecise constraints we use the example shown in Fig. 6. First, in Line 1 the option *HOST* is accessed, which is of type string, which LOTRACK currently cannot track precisely. Instead, a taint for variable `url` is created indicating that this variable is related to the option *HOST*. In Line 2, the if-statement's condition includes the tainted variable `url`, because of the imprecise

tracking the resulting constraint is $HOST_\alpha$. Line 3 is only reached under the imprecise constraint $HOST_\alpha$ which becomes the result of the configuration map for this line. The corresponding Line 5 of the *else* branch has the constraint $\neg HOST_\alpha$. Therefore, we can state that this line is *somehow* related to the option but not exactly how. The constraint related to *HOST* is resolved after the if-statement since $HOST_\alpha \vee \neg HOST_\alpha = true$

Second, we show in Lines 6 and 7 how LOTRACK will fall back on imprecise tracking when value tracking is no longer possible. In Line 6, an integer option is accessed for which value tracking is possible. Then, the tainted variable is passed to a function with a second parameter whose values cannot be determined statically. LOTRACK cannot calculate how the resulting variable `a` is related to the originally accessed option, it falls back on imprecise tracking for variable `a`. As an overapproximation, LOTRACK assumes `a` is related to the option *Version* but will not make any assertions about its value. The use of the imprecisely tracked variable in Line 8 will result in the constraint SDK_α.

LOTRACK seamlessly combines precise and imprecise constraints. We illustrate this using the small example in Fig. 7. First, the use of the *SDK* option is similar to the example from Fig. 4 and can be tracked precisely. The access of the string option *Model* (Line 3) will taint the variable `urlPart` with the imprecise value $MODEL_?$, i.e., its value is unknown but related to the option *MODEL*. The taint's constraint is SDK>8 which is the condition that Line 3 is even reached. The taint for `urlPart` is used in an if condition (Line 4) for which LOTRACK creates the imprecise constraint $MODEL_\alpha$ because the tool only knows this variable is related to *MODEL* but does not understand the condition. To create the final result for the configuration map, the constraint for the condition is combined with the constraint of the taint resulting in constraint $SDK > 8 \wedge MODEL_\alpha$ for Lines 5 to 6. The assignment of a constant value to the variable `lvl` within the if branch will create a *value taint* with the constant value 5 and the current constraint ($SDK > 8 \wedge MODEL_\alpha$) as the taint constraint (Line 5). The value of this taint will be tracked precisely even though the constraint contains imprecise parts.

3.3 Formalization

As described before, the rules for creating and propagating taints are nuanced and depend on various conditions, such as whether precise information can be tracked in taints at a given part of a program. Especially the computation of constraints, which are then propagated through taints, is nontrivial. To systematically describe how we create constraints, we formulate a set of rules. The rules are expressed over options, option values, taint values, and constraints.

A program has a set of configuration options $\mathbb{O}$. Each configuration option $o \in \mathbb{O}$ has a domain, $dom(o)$, that describes all possible concrete configuration values for that option (e.g., $dom(Debug) = \{true, false\}$). Precise value tracking is only possible for options with finite domains. A symbolic value $sym(o)$ for each option o represents the selected configuration of that option. In addition, to enable imprecise tracking in constraints, we introduce additional symbolic configuration values for each option, e.g., o_α, o_β for an option o (infinite set $imp(o)$ for each option o); function $freshSym(o)$ produces a fresh symbolic configuration value for an option o.

A taint $t = (w, v, c)$ is a tuple consisting of a variable w in the program under analysis, a corresponding taint value v, and a constraint c. A taint value is either a concrete value from the set of all concrete program values $\mathbb{A}$ (e.g., $true$, 1), a symbolic value of a configuration option, a symbolic option representing an unknown value related to an option, or an entirely unknown value $\perp$: $(v \in \mathbb{A} \cup \left(\bigcup_{o \in \mathbb{O}} imp(o) \cup \{sym(o)\}\right) \cup \{\perp\})$. A constraint is a formula over taint values:

$$c ::= v_1 > v_2 | v_1 < v_2 | v_1 = v_2 | c_1 \wedge c_2 | c_1 \vee c_2 | \neg c | true$$

At each point in the program, we track a potentially large number of taints Π. A variable may be tainted by multiple taints. As shorthand, we use the notation T_x to denote all taints for a variable x at a specific point in the program with taints Π:

$$T_x = \{(w, v, c) \in \Pi \mid w = x\}$$

3.3.1 Normalization of taint sets

If a taint set describes the value of a variable for all possible configurations in the current context, we call it *complete*. It is common to have incomplete information at some point during the analysis, in which we only know the value of a variable for some configurations but not for others. To simplify and unify the subsequent constraint computation, we first normalize incomplete taint sets by introducing artificial taints representing special *unknown* values for configurations not yet covered.

Completeness is determined based on the constraints in a taint set. To that end, function ϕ builds a disjunction of all constraints in a taint set T as[4]

$$\phi(T) = \bigvee_{(\_,\_,c) \in T} c$$

A taint set T is complete, iff $\phi(T)$ holds for all configurations of the current context ψ, i.e., the current configuration constraint from surrounding if-statements—that is, if $\psi \Rightarrow \phi(T)$. We illustrate this check using the example shown in Fig. 8: In Line 2, we have obviously complete information since $\phi(T_a) = true$; in Line 3, $\phi(T_a) = A$ but since the if-statement's constraints, as shown in the configuration map left to the statement, is also A we have complete information regarding variable `a` as well. Naturally, an empty taint set ($T_w = \emptyset$) is always incomplete in any reachable context because it provides no information about any configuration.

To normalize a taint set, we complement it, if incomplete, with a synthesized taint to explicitly model any missing information. This synthesized taint will later trigger the creation of imprecise constraints to indicate uncertainty. A normalized set is always complete, so we will not have to deal with incomplete sets during constraint computation, only with unknown information within complete sets. Specifically, we add a taint with a special *unknown* value $\perp$ and the constraint that represents all configurations not yet covered by the taint set: $\neg\phi(T)$. We use the notation T_x^n to

4. As usual, we define the disjunction over an empty set as *false*, i.e., $\phi(\emptyset) = false$.

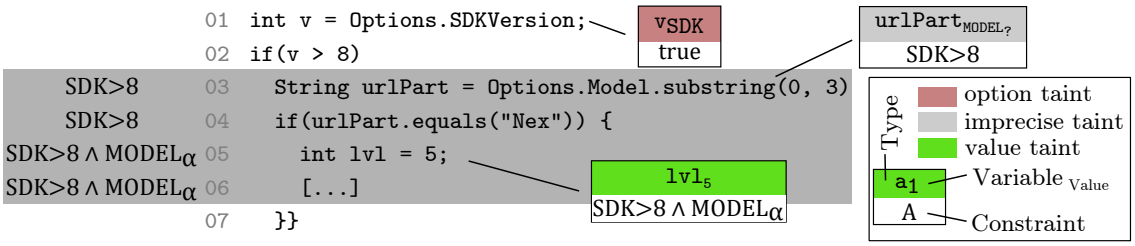


Fig. 7: Combination of precise and imprecise constraints.

	Example	Taints	Disjunction	Complete
	01 int a = Option.A ? 1 : 0;			
	02 if(a == 1)	{(a,1,A),(a,0,¬A)}	$\phi(T_a)$=true	true
A	03 if(a > rand())	{(a,1,A)}	$\phi(T_a)$=A	true

Fig. 8: Check for *complete* information for variable a.

represent a normalized taint set for a variable x at a given position in the program:

$$T_x^n = T_x \cup \{(x, \bot, \neg\phi(T_x))\}$$

3.3.2 Constraint Creation for Branching Decisions

At each branching decision in the control-flow graph, we need to identify a constraint under which the subsequent code is executed. This constraint will be used to restrict the constraints of taints that are propagated to the respective branches. The difficult part is to track information precisely as long as possible, but to fall back on imprecise tracking where unavoidable.

Without loss of generality, we assume that all control-flow decisions are reduced to if-statements comparing values of two variables *if (x ⊕ y) ...* in which ⊕ stands for a comparison operator $<, >, \leq, \geq$, or $=$. This form is close to the Jimple format underlying our implementation (Section 4); other control-flow decisions can be reduced to if-statements; comparisons of other constructs than variables can be encoded by storing their values in temporary variables first; unary if conditions can be encoded (e.g., *if (x)* as *y=true; if (x==y)*).

If we know all possible values for variables x and y for all configurations, we can create a precise constraint. In contrast, if we have no information neither for x nor y we return the trivial constraint (*true*), assuming there is no relation of the branching condition to configuration values. In all other cases, we have *incomplete* information, i.e., we know a variable's values only for certain configurations, which forces us to use imprecise constraints.

For every pair of taints for x and y, we create a new constraint comparing the values of those constraints, combined with the prior constraints of those taints. Specifically, we compute the constraint based on the normalized taint sets T_x^n and T_y^n for variables x and y at the point of the control flow decisions, in which x and y are compared with operator ⊕ as $c(T_x^n, T_y^n, \oplus)$, which we show in Fig. 9.

There are different combinations of available information in the two taint sets: If we have no information about either value, we simply do not create stricter constraints than the conditions of the taints. If we know both values, we use ⊕ to translate an operator from the if-statement to an operator as part of a constraint between two values, thus expressing the condition among the variables within the constraint. If either value is unknown though, we need to fall back on imprecise handling: We use a special function *imprecise* that takes a value or constraint and returns a constraint referencing all occurring configuration options with fresh symbolic configuration values for those options.[5] Essentially, this function allows us to create an imprecise variant of any information related to an option encoded in value v and constraint c.

3.3.3 Examples

To illustrate the rules for constraint creation, we will now present a set of examples of how our constraint computation c is applied to different scenarios of available taints:

1) **Comparison of two tainted variables with complete information:** If both variables are tainted with concrete values under constraint *true* (i.e., in all configurations), we compute a constraint that is either true or false, as in the following example comparing two constants:

$$\begin{aligned} &c(\{(x, 1, true)\}, \{(y, 5, true)\}, <) \\ &\quad = (1 < 5) \wedge true \wedge true = true \end{aligned}$$

If we track different concrete values under different conditions, we compute the resulting constraints based on those conditions, as in the following example in which we track different concrete values for x depending on option A:

$$\begin{aligned} &c(\{(x, 1, A), (x, 6, \neg A)\}, \{(y, 5, true)\}, <) \\ &\quad = ((1 < 5) \wedge A \wedge true) \vee ((6 < 5) \wedge \neg A \wedge true) \\ &\quad = A \vee false \\ &\quad = A \end{aligned}$$

5. Technically, *imprecise*(v) works as follows: Given a constant value ($v \in \mathbb{A}$) it returns *true*; given a concrete or symbolic configuration option ($\exists o.v \in imp(o) \cup \{sym(o)\}$), it returns a fresh symbolic configuration value for that option (*freshSym*(o)). When applied to constraints (*imprecise*(c)), it recursively applies *imprecise* to every value within that constraint and returns a conjunction of the results, thus capturing all options mentioned anywhere in the constraint.

$$c(T_x^n, T_y^n, \oplus) = \bigvee_{\substack{(\_,v_x,c_x)\in T_x^n \\ (\_,v_y,c_y)\in T_y^n}} \begin{cases} (v_x \oplus v_y) \wedge c_x \wedge c_y & \text{if } v_x \neq \bot \wedge v_y \neq \bot \\ \mathit{imprecise}(v_y) \wedge \mathit{imprecise}(c_x \wedge c_y) & \text{if } v_x = \bot \\ \mathit{imprecise}(v_x) \wedge \mathit{imprecise}(c_x \wedge c_y) & \text{if } v_y = \bot \\ \mathit{imprecise}(c_x \wedge c_y) & \text{if } v_x = \bot \wedge v_y = \bot \end{cases}$$

Fig. 9: Definition of formula $c(T_x^n, T_y^n, \oplus)$ to calculate a constraint from a set of taints.

Finally, the same computation also works for symbolic values, for example, when we know that x represents the configuration value of option A:

$$\begin{aligned} &c(\{(x, \mathit{sym}(A), true)\}, \{(y, 5, true)\}, <) \\ &\quad = (\mathit{sym}(A) < 5) \wedge true \wedge true \\ &\quad = \mathit{sym}(A) < 5 \end{aligned}$$

2) **Comparison of a tainted with an untainted variable:** If variable y is untainted, we need to fall back on imprecise tracking, because we have no information about the possible values of y. The case for an untainted variable x and a tainted variable y is symmetric. We already show the normalized input with a $\bot$ value for the empty taint set T_y.

$$\begin{aligned} &c(\{(x, 1, true)\}, \{(y, \bot, true)\}, <) \\ &\quad = \mathit{imprecise}(1) \wedge \mathit{imprecise}(true \wedge true) \\ &\quad = true \wedge true = true \end{aligned}$$

$$\begin{aligned} &c(\{(x, 1, A), (x, 0, \neg A)\}, \{(y, \bot, true)\}, =) \\ &\quad = (\mathit{imprecise}(1) \wedge \mathit{imprecise}(A \wedge true)) \vee \\ &\qquad (\mathit{imprecise}(0) \wedge \mathit{imprecise}(\neg A \wedge true)) \\ &\quad = A_\alpha \vee A_\beta \end{aligned}$$

The resulting constraint $A_\alpha \vee A_\beta$ represents both the relation to the option A, that we know from the taints for variable `x`, but the imprecise constraint also indicates our lack of knowledge about the second operator `y`. Following the steps in the formula, we can see how the two incoming taints for variable `x` lead to the two imprecise symbols A_α and A_β.

3) **Incomplete taint information:** Consider variable x with a single taint $(x, 1, A)$ representing only incomplete information. During normalization, we complement it, leading to the following case:

$$\begin{aligned} &c(\{(x, 1, A), (x, \bot, \neg A)\}, \{(y, 5, true)\}, <) \\ &\quad = \big((1 < 5) \wedge A \wedge true\big) \vee \\ &\qquad (\mathit{imprecise}(5) \wedge \mathit{imprecise}(\neg A \wedge true)) \\ &\quad = A \vee A_\alpha \end{aligned}$$

The second part of the result, A_α, may not be intuitive since the condition $x \oplus y$ could also be satisfied by a value with no relation to option A. As we have explained in the introduction of imprecise constraints (Section 3.2.5), we always enforce a relation to an option in an imprecise symbol to model existing information. The interpretation of A_α therefore also allows only a *potential* association to the option.

4) **No taints:** If there are no taints for a condition, the resulting constraint will always be *true*:

$$\begin{aligned} &c(\{(x, \bot, true)\}, \{(y, \bot, true)\}, <) \\ &\quad = \mathit{imprecise}(true \wedge true) = true \end{aligned}$$

3.3.4 Algorithm

We continue the formalization of our approach by integrating the rules to create constraints at branching decision into the taint analysis algorithm. LOTRACK works on top of a taint analysis which provides the functionality of taint creation and propagation as well as common features of static program analysis like call-graph creation and alias analysis. Besides a basic overview of the algorithm for taint analysis, we concentrate on the extension for constraints and refer for a more detailed description of the basic taint-tracking mechanisms to the work on FLOWDROID [4].

Our value-based taint-tracking algorithm shown in Fig. 10 requires an inter-procedural control-flow graph and an initial edge as input. To handle multiple possible entry points of a program, which is common for Android apps as well as some types of Java applications, the underlying FLOWDROID tool creates an artificial main method, which calls every possible entry point. The main method also simulates the initialization of static class members.

The analysis works at the level of inter and intra-procedural control-flow edges. An edge consists of source and target taint values, the source and target statements, as well as a constraint.

The algorithm is initialized by adding the initial edge with constraint `true`, given by the control-flow graph, to the set of edges to be processed (Line 2). More edges will be created from the algorithm itself as it traverses the control-flow graph.

For each edge, the following basic steps are taken. The successors of the edge's target statement are determined using the inter-procedural control-flow graph (Line 6). Using a normal taint analysis, the possible taints at the successor are determined based on the current edge (Line 7). New taints have a constraint c initialized with *true*.

The constraint for each taint at the successor is determined using the rules presented before (Line 8). Note that *createConstraint* (Section 3.3.2) will already conjoin the constraint propagated to this edge with the new constraint for this edge. For example, if a taint has the constraint $\neg A$ and the constraint from the control-flow edge is B, the resulting constraint for the taint will be $\neg A \wedge B$. As an optimization, taints with an *unsatisfiable* constraint do not need to be propagated further.

Along different paths, different taints with the same taint information (i.e., the same variable and value) can reach the same statement. In this case, the taints are joined to a

input : inter-procedural control-flow graph (icfg), initial edge (points to the first statement of the program and contains a dummy taint without any relation to a configuration option)
output: map of taints for each statement

```
1  Function trackTaints
2    edges ← {initialEdge};
3    result ← {};
4    while edges is not empty do
5      extract edge ⟨sourceStatement, targetStatement, sourceTaint⟩ from edges;
6      for successor ∈ getSuccessors(icfg, targetStatement) do
7        for taint ⟨w, v, c⟩ ∈ computeTaints(successor, targetStatement, sourceTaint) do
8          constraint ← c ∧ createConstraint(successor);
9          constraint ← constraint ∨ existingConstraint(successor, taint) ;
10         taint ← ⟨w, v, constraint⟩;
11         if (taint ∉ results[successor]) ∨ (constraint ⊐ existingConstraint(successor, taint)) then
12           results ← results ∪ {successor ↦ taint};
13           edges ← edges ∪ ⟨targetStatement, successor, taint⟩ ;
14         end
15       end
16     end
17   end
18   return result;
19 end
```

Fig. 10: Taint-tracking algorithm

single fact disjoining the individual constraints (Line 9). The algorithm uses the function *existingConstraint(s, t)* to retrieve the current constraint of the taint *t* at statement *s*, which will return *false* if there is no previous value. The function will return *false*, if there is currently no constraint for the values s and t.

The taint produced by the underlying taint analysis is updated with the final constraint (Line 10). The loop finishes with a check whether the created taint will extend the current result set and as consequence needs to trigger further propagation (Line 11). A taint for a previously untainted variable or a taint with a new value will always be added to the result set. A taint which only differs in its constraint will be added to the result set, if the new constraint is more relaxed than the existing constraint. We check this based on the *partial order* of constraints which we discuss in detail shortly. The taint with the final constraint together with the successor statement results in a new edge which is added to the list of edges to be processed (Line 13). The algorithm finishes once there are no more edges to process.

With this check we can make the argument for termination of the algorithm. First, the control-flow and data-flow graphs only contain finite numbers of statements and variables. Second, we use value tracking only for options with finite domains, i.e., Boolean and integer options (Section 3.2). This already restricts the number of results with respect to the statements as well as tainted variables and taint values. The taint constraint is initialized using the function *createConstraint* and disjoined with the existing constraint. As a consequence, with each iteration the constraint of a taint can only become more relaxed and the disjunction will eventually stop at *true*. In the context of imprecise constraints, we define a *partial order* of constraints to define when a constraint is *more relaxed* thus justifying further propagation, for this case we use the notation $c_1 \sqsupset c_2$.

The intuition behind the partial order of constraints is that we want order constraints based on the number of possible configurations they represent. The mix of precise and imprecise symbols as well as the way we handle negations in imprecise constraints makes the definition of the order difficult.

We base the partial order on *logical implication*. We do not want to consider any operation between imprecise symbols for the same option to make the result more or less restrictive, e.g. $A_\alpha \vee A_\beta$ provides no more information than A_β whereas $A \vee B$ is indeed a more informative result than B. To determine the order of a set of formulas, we first create the conjunctive normal form (CNF) of the formulas and substitute any imprecise symbol $v \in imp(o)$ or its negation $\neg v | v \in imp(o)$ with the symbol $\bar{o}$. The CNF ensures the following substitution will not remove any relevant symbols. We demonstrate the substitution using the following two examples:

$$\begin{aligned} A_\alpha \vee B_\alpha \vee A_\beta &\xrightarrow{substitute} \bar{A} \vee \bar{B} \\ (\neg A_\alpha \wedge B) \vee A_\beta &\xrightarrow{CNF} (\neg A_\alpha \vee A_\beta) \wedge (B \vee A_\beta) \\ &\xrightarrow{substitute} (\bar{A} \vee \bar{A}) \wedge (B \vee \bar{A}) \end{aligned}$$

Based on this normalized form, we define the partial order for formulas with precise symbols, imprecise symbols and any combination using implication. We illustrate the complete process to determine the order of two constraints using a small example. We want to check the order of two formulas:

$$\begin{aligned} l_1 = A_\alpha \wedge A \quad \text{and} \quad l_2 &= A_\alpha \wedge A_\beta \\ l_1' = \bar{A} \wedge A \quad \text{and} \quad l_2' &= \bar{A} \qquad \text{(normalize)} \\ \bar{A} \wedge A \implies \bar{A} &\qquad \text{(check implication)} \\ A_\alpha \wedge A < A_\alpha \wedge A_\beta &\qquad \text{(resulting order)} \end{aligned}$$

3.4 Example

To illustrate and summarize the complete approach, we walk through a nontrivial example shown in Fig. 11. On the

```
01 public void start() {
02   boolean z = isConfig();
03   if(z)
04     method1();
05   z = false;
06   boolean c = hasC();
07   if(c)
08     boolean e = hasE();
09     return;
10   c = true;
11   int d = getD();
12   if(d > rand())
13     if(d < rand())
14       return;
15   method2();
16 }
17
18 public boolean isConfig() {
19   //return hasA() && !hasB()
20   boolean a = hasA();
21   if(a) {
22     boolean notB = hasB();
23     if(notB)
24       return true;
25   }
26   return false;
28 }
```

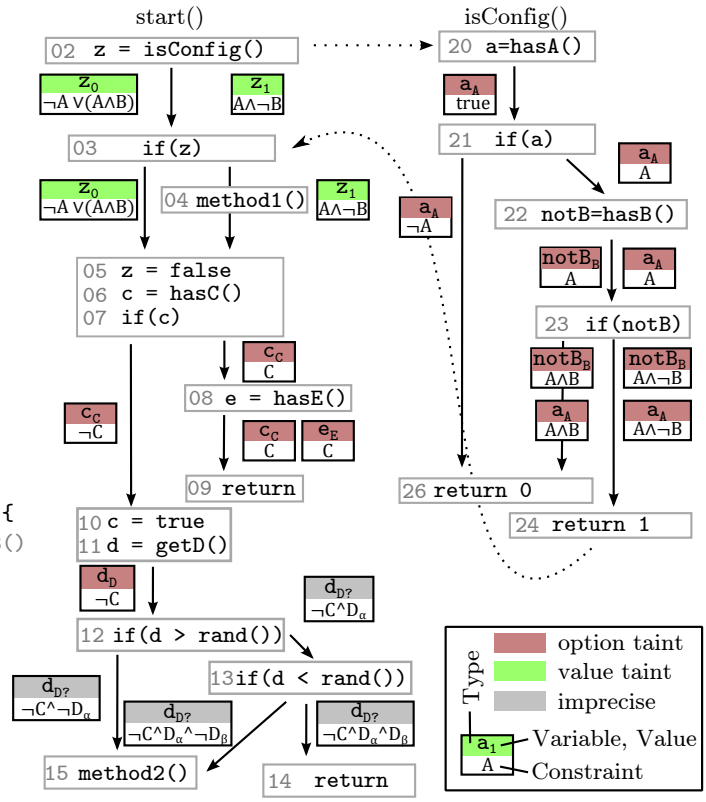


Fig. 11: Source code and corresponding call-graph annotated with tracked taints

TABLE 1: Configuration API and Value Tracking Information for Example (Fig. 11)

API	Option	Value Tracking
`com.company.hasA()`	A	[+,-]
`com.company.hasB()`	B	[+,-]
`com.company.hasC()`	C	[+,-]
`com.company.getD()`	D	$\mathbb{N}$
`com.company.hasE()`	E	[+,-]

left side, we show Java source code of two simple methods. On the right side, we show a control-flow graph annotated with information regarding the data-flow information being tracked through the program.

The shown source code may seem verbose but this format is closer to the representation used by LOTRACK internally and makes it easier to follow the analysis steps. This does not affect the ability to handle the full set of Java. For example, Lines 20-26 could be written equivalently as shown in the comment in Line 19. We will discuss the implementation aspects of LOTRACK in more detail in Section 4.

Our analysis proceeds as follows. First, we need the information which API can be used to retrieve the value of configuration options and whether value tracking is used. Table 1 shows the necessary input for our example.

Next, we start the actual taint analysis with the entry point, in this case, the edge calling method `start`, from where we analyze its first statement, the call to method `isConfig`. Before continuing within `start`, we need to handle the called method to identify all possibly relevant results. The first statement of `isConfig` is the call to `hasA`, to retrieve the configuration option A (see Table 1). LOTRACK creates the single option taint $(a, A, true) = \Pi$. As an option taint, it represents any possible configuration values of A, which in this case are *true* or *false*. The taints constraint is *true*, because this taint is created independent from any configuration. Fig. 11 shows taints as boxes.

At the following if-statement (Line 21), the constraint calculation rules (Section 3.2) are used to create constraints for both outgoing edges (branch and fall-through edge) at this statement based on the condition (`a = true`) and the only available taint. The resulting constraints (A for branch edge and $\neg A$ for fall-through edge) are applied to the taint resulting in the taint (a, A, A) being propagated to Line 22 and taint $(a, A, \neg A)$ being propagated to Line 26.

In Line 22, the call to method `hasB()` is used to access the configuration option B and immediately assign the Boolean result to variable `notB`, resulting in a new option taint. The new taint is only propagated after its constraint is conjoined with the existing constraint of this branch resulting in $(notB, B, A)$ at Line 23. Taint (a, A, A) is propagated unchanged.

At the two possible return points (Lines 24 and 26), the new taints for the variable `z` at the call site are derived from the taints in the called method. The statements return literal values (0 or 1), therefore LOTRACK creates value taints for variable `z`. At Line 26, for instance, the constraint for $(z, 0, \_)$ is calculated from the three incoming taints as $\neg A \vee (A \wedge B) \vee (A \wedge B)$, simplified to $\neg A \vee (A \wedge B)$.

Back in method `start` both taints for variable `z` are propagated from Line 2 to 3. The if-statement at Line 3 is

handled the same way as previously described. At Line 5, the value of `z` is overwritten by a non-configuration value, which is why taints for `z` are not propagated further along.

After the creation of the constraint on option C (Line 6) and its use (Line 7), Line 8 will depend on C. In Line 8 the option taint (e, E, C) is created but without any influence on the resulting constraint because the variable is not used in any control flow decision. Because `start()` returns no value, the analysis will not propagate any taints beyond the return statement at Line 9. The only taint at Line 10 has the constraint $\neg C$, which is the final result for this statement.

The handling of option D is different from the previous example because it is used in a way which forces the analysis to fall back on imprecise configuration handling. First, the option taint $(d, D, \neg C)$ is created at Line 11 and propagated to Line 12. Here, the tainted variable is compared to a random value (return by `rand()`), which cannot be resolved statically. Instead, the imprecise constraint D_α is created to indicate that the condition is related to option D but is it unknown *how*. The taints $(d, D_?, \neg C \wedge \neg D_\alpha)$ and $(d, D_?, \neg C \wedge D_\alpha)$ are propagated along the branching edge and the fall-through edge, respectively. A similar condition is found again at Line 13. To be able to differentiate between different imprecise constraints another placeholder symbol is introduced, D_β. Although the creation of different symbols at each condition with is handled imprecisely increases the complexity of the resulting constraints, it allows LOTRACK to merge the constraints correctly. The result in the configuration map for Line 15 is, based on the two taints, $(\neg C \wedge \neg D_\alpha) \vee (\neg C \wedge D_\alpha \wedge \neg D_\beta)$, which can be simplified to $\neg C \wedge (\neg D_\alpha \vee \neg D_\beta)$.

4 IMPLEMENTATION

LOTRACK implements the approach presented in Section 3 for Java applications and for Android apps. The implementation is based on FLOWDROID [4], a tool for taint analysis, and SPL$^{\text{LIFT}}$ [9], both are in turn based on Soot [54]. FLOWDROID specifically supports a taint analysis of Android apps by, e.g., automatically detecting all possible entry points to an app. Soot is a framework for implementing Java analyses. Input files, i.e. Java source code or Android bytecode, are transformed to the intermediate *Jimple* format. The Jimple format supports analyses by, e.g., transforming complex expression to a set of simpler expressions and introducing variables holding provisional results.

The necessary information on relevant API calls and configuration options are given in a simple configuration file, which makes it easy to adapt for most software systems. For all operations related to constraints we use the SMT solver Z3 [32]. This solver allows us to support operations on Boolean and integer options. String options are currently not supported but the tool could be adapted for these as well using a corresponding solver such as Z3-str2 [57]. [6]

6. Currently, we translate Java expression on Boolean and integer types to expression for the solver which directly supports them. An extensions like Z3-str2 supports additional operations in the solver for strings. Therefore, we would need to translate Java string operations, e.g. substring or string length, to corresponding expressions in the string solver to determine the satisfiability of conditions and to build constraints.

Our formalization (Section 3.3) expects taints for constant values used in branching statements. In general, we do not track every constant but only those that have an indirect dependence on configuration values. Also, we do not expect developers to use constant literals directly since the use of such *magic constants* is considered bad practice. Instead, we make use of the abstraction provided by the Jimple intermediate form and perform constant propagation prior to the analysis which will resolve constant values used in branching statement.

To make it easier to use the analysis results, LOTRACK displays the extracted constraints within the original Java code instead of the intermediate Jimple code. The mapping of Jimple to Java code lines is possible if the compiler is set to include line numbers in the resulting bytecode files, which is a common debug setting. This enables integrating LOTRACK into IDEs.

The current implementation has limitations which can both lead to missed constraints as well as an overapproximation of constraints. For instance, dynamic binding of function calls is currently handled such that taints are propagated to all possible target methods. To overcome this limitation, we would need a points-to analysis that can provide correct binding depending on the configuration constraints. Overapproximation can also happen due to unknown implementation of functions (e.g., in native libraries). As we will show in our evaluation though, LOTRACK achieves high accuracy.

5 EVALUATION

Toward our goal of providing developers with practical tool support that can recover a configuration map for different maintenance tasks, we evaluate LOTRACK on real-world applications.

There are different challenges associated with our approach of creating a configuration map:

1) Accuracy: To gain valid insights into the analyzed application the results need to be accurate. Especially with imprecise constraints, it is not obvious what an *accurate* configuration should look like. We expect precise constraints where this is possible given the limitations of the tool. The accuracy of imprecise constraints is a challenge, because imprecise constraints contain generated symbols which a developer cannot be expected to manually recreate. To simplify the assessment, we will count imprecise constraints as correct if the effect of the options mentioned in the constraint is traceable. Although there are known cases where the results will be incomplete, we expect a high accuracy for real applications.
2) Performance: The performance should enable the analysis of real-world applications. Especially an approach like value tracking will likely lead to performance problems. However, due to our focus on configuration values and our experience on how these options are commonly used, we expect our approach to scale beyond trivial applications.
3) Usefulness: The resulting configuration maps need to be a relevant basis to support software engineering tasks or further analyses.

TABLE 2: Android Configuration Options (excerpt)

API	Option
`android.os.Build$VERSION:int SDK_INT`	SDK
`Configuration.locale`	LOCALE
`Environment.getExternalStorageState()`	STORAGE
`Context.getSystemService("vibrator")`	VIBRATOR
`Context.getSystemService("bluetooth")`	BLUETOOTH

The goal of our evaluation is to check how our approach as well as our implementation can address these challenges We first evaluate the *accuracy* of our recovered configuration maps in terms of precision and recall. Second, we evaluate the *performance* of our analysis using a set of Android apps and Java applications. Finally, we indirectly demonstrate *usefulness* by performing an empirical study on how configuration options are used within tested systems and how configuration options interact. Our evaluation has two parallels thrusts: we both perform the evaluation on a set of Android apps as well as command-line Java applications.

5.1 Subject Systems

5.1.1 Android Apps

Android apps are an interesting subject for tracking configuration options, because the Android platform provides many configuration options, up to the point that the Android platform has gained a reputation for fragmentation into many different hardware and software versions and variants. Android apps use load-time options to determine the availability of software and hardware functionality at runtime. Configuration options are accessed through standard APIs, which means that we can study many apps with the same configuration options without the overhead of identifying each system's configuration options separately. Furthermore, there is a large research community that has already prepared tool chains for analyzing Android apps that we can build on. Finally, there are a large number of free and open-source apps available to study.

As subjects for our evaluation, we randomly selected 100 apps from the FDroid[7] repository of open-source Android apps. The first 10 of these 100 apps are selected for a more detailed analysis as part of our evaluation regarding accuracy (Section 5.3). These selected apps, including their size, are shown in Table 3.

5.1.2 Java Applications

The analysis of generic Java applications is an important part of the evaluation because it shows that the approach can be used for applications that use configuration options in different ways and differ in size and domain. Unlike Android apps, Java application, as used in our evaluation, pose the challenge of having no common framework to define or access configuration options.

We used the following process to select applications: First, we determined criteria to search for applications that are relevant for our analysis. From our experience, a strong indicator for projects having load-time configuration options is the presence of a file named *commandlineargs.java*. We used GitHub's search engine to find matching projects.[8] From the query results, several inapplicable projects were excluded because they could not be built or are frameworks that have no entry point or contain no configuration options. We selected the first 10 applicable applications. We only take a small number of Java applications because the effort for their analysis in the evaluation is higher compared to the analysis of apps. For each application, we need to build it and setup the analysis (Section 5.2).

Table 4 shows the Java software systems and some key properties of them. The first 4 of these 10 applications are selected for a more detailed analysis regarding accuracy.

5.2 Setup

LOTRACK requires two steps in its setup: a list of configuration options along with information on the API to access them and a list of entry points, which is used to start the analysis.

5.2.1 Android apps

We make use of the Android framework to setup configuration options and detection of entry points once and reuse this setup for every Android app we analyze.

Experts on the Android framework provided us with a list of classes commonly used for configuration purposes [39], which we used to build the set of configuration options. In total, we selected 51 options from the Android documentation, including a wide array of different options regarding hardware and software (e.g., availability of SD card, usable sensors, or framework version).

For each of these options, we identified the API for reading the configuration value. We use precise value tracking for all integer and Boolean options (20 of all 51 configuration options). In Table 2, we show an excerpt of the identified options; a full list is available on LOTRACK's web page. Note that the list of configuration options could be easily changed or extended to include, e.g., app-specific or user-defined configuration options.

The entry points define at which statements the construction of the control-flow graph starts. For Android apps, we reused the implementation of FLOWDROID, to recover entry points based on an app's intents, callbacks, etc. LOTRACK allows the declaration of individual entry points, even multiple entry points.

5.2.2 Java Applications

In case of Java, we have to setup the tool for each individual application because we can no longer assume a standardized framework.

In the Java ecosystem, there is no single configuration system, in some cases even within a single application [38], which would enable a complete and consistent analysis of the configuration options for *all* Java applications. There are numerous examples of libraries for reading configuration files, parsers for command-line options or simply cases

7. https://f-droid.org

8. We used the following query on 2015-10-28: https://github.com/search?p=1&q=commandlineargs.java+in%3Apath&ref=searchresults&type=Code. The links to the selected project repositories can be found at https://gist.github.com/MaxLillack/dc6255fb3cc5a48d8ba5.

TABLE 3: Comparison of LOTRACK's Results and Manually Created Oracles on Ten Apps and Four Applications

		correct		wrong	
Name	Size (Java LOC)	like oracle	better than oracle	missed	overapproximation
Import Contacts	3,570	4	7	0	0
Nectroid	4,724	4	4	0	2
OSChina	23,280	5	14	4	0
Tinfoil for Facebook	1,364	6	3	0	2
AnySoftKeyboard	18,873	2	3	1	3
Mounts2SD	3,618	2	0	1	0
Impeller	7,389	1	0	0	0
KeePass NFC	1375	7	2	1	3
Dolphin Emulator	1,812	1	0	1	0
Document Viewer	50,317	15	9	4	12
platypus	20,284	8	6	0	1
kafka-dispatch	175	6	1	0	0
Data Consumer	1,677	9	0	0	0
AndSync	883	2	0	3	0
Sum		72	49	15	23

TABLE 4: Java Applications Used in Evaluation

Name	Description	#Options	LOC
platypus	Page Layout and Typesetting Software	9	20,284
kafka-dispatch	Addon for Apache Kafka	7	175
Data Consumer	Academic data analysis	8	1,677
AndSync	Synchronization library	7	883
ProteaJ	Compiler	6	9,818
adligo	Build system	7	20,577
RemoteREngine	R package	5	3,188
M-Grid	Big Data indexing and querying	54	27,095
jmxetric	JVM instrumentation	7	1,546
WarGameofThrones	Game	9	8,836

where configuration options are implemented ad hoc. We manually defined the necessary configuration APIs based on the documentation and the source code of the applications, techniques to automatically detect configuration options [38] can complement our approach.

To determine the entry points for Java applications, we simply use the *main* method.

5.3 Accuracy

Before we use our tool to study configuration options in practice, we first evaluate its accuracy. To obtain an oracle, we manually created a configuration map for the subject systems. Subsequently, we automatically extracted a configuration map with LOTRACK and compared it to the manual result, yielding measures of precision and recall.

5.3.1 Oracles

We are unaware of any Java applications or Android apps in which the mapping of code fragments to configuration constraints has been explicitly documented so that we could use them as oracle for our study. To establish ground truth, we manually investigated a set of sample systems, creating oracles.

To create oracles, we first documented the process that a human developer would take to track configuration options and to create a configuration map. This document includes the configuration options and corresponding API calls that should be tracked and a list of possible entry points.

For every subject system, we asked at least two experts (at least one author and at least one researcher not involved in this project) to independently identify and track all configuration options in the Java source code of the system with the goal of describing all code fragments that are triggered by the configuration options. All experts have multiple years of experience in Java and the used IDE. The experts discussed all differences in their results with the goal of either unanimously agreeing on a correct version or clarifying the process documentation. In fact, we found that the process documentation was clear enough and that all differences could be explained by omissions by one expert, which occurred a few times in larger applications. In fact, our experience in creating the oracles anecdotally confirms that creating configuration maps is well defined but tedious and error prone when performed manually. The experts needed up to 30 minutes per system. Using search features of IDEs, the access of configuration APIs can be identified easily, but one quickly loses track of the use of the accessed configuration values and their sometimes extensive impact, e.g., on called methods.

To evaluate accuracy, we compare the configuration map automatically derived by our tool (from the bytecode files) with the manually derived oracle. We count continuous lines of Jimple code with the same constraints only once to prevent a bias towards uses of configuration options that affect a large number of lines. We measure recall as Jimple code that is correctly mapped to a configuration constraint compared to Jimple code that is mapped to some configuration constraint in the oracle. We measure precision as Jimple code that is correctly mapped to a configuration constraint compared to Jimple code that is mapped to some

configuration constraint by our tool. A correct mapping requires the exact identification of the affected statements as well as the correct constraint, for imprecise constraints we only expect that the correct options are referenced in the constraint.

5.3.2 Results

In Table 3, we show the observed accuracy of LOTRACK's results: LOTRACK reaches a precision of 84% and a recall of 89%. There was no case of incorrectly detected constraints: the constraints were either correct or missed entirely. There were no cases constraints that were detected by LOTRACK which did not match the expected constraint. However, the results contain cases of *missed* constraint, i.e., a constraint was expected but was not part of the configuration map, and cases of *overapproximation*, i.e., the configuration map contained constraints which were not expected.

In most cases, LOTRACK's result agrees with the oracle. In 49 cases the tool identified constraints for lines that were missed by all experts when creating the oracle. Checking back with our process instructions, we could confirm that the tool was correct and the experts were wrong. This occurred especially for indirect dependencies and methods called only from optional code.

LOTRACK missed valid constraints (15 cases) mostly due to an incomplete call graph. For instance, some callbacks from the framework were unknown and therefore not handled by the underlying FLOWDROID implementation.

Overapproximation occurred for 23 pieces of code, where most of the cases seem to be related to overly approximate points-to analysis, a well-known problem that all static analyses share.

Overall, our results indicate that the analysis is highly accurate. In a few cases it has even corrected developers carefully performing the task manually to build the oracle, and overapproximation had only a minor effect.

5.4 Performance

To ensure practicality, we evaluate performance in terms of analysis time and memory consumption. We report the median wall-clock time as reported by *JUnitBenchmarks*[9] of five runs after three discarded warm-up runs, on a Core i7 notebook with 3.3Ghz and 16 GB memory. Warm-up runs and repeated measurements are established techniques to reduce the impact from JVM self-optimization and concurrently running applications. The number of warm-up runs and measurement runs is a compromise between measurement accuracy and required time to run the experiment.

For memory consumption, we report the peak memory usage. We automatically performed the analysis on the 10 apps from our accuracy analysis and 90 additional randomly sampled apps from the FDroid repository ranging from 17 to 82,000 lines of Jimple code, listed on the project's web page. The median time for the analysis is 8 seconds; the longest runtime was 47 minutes. All apps could be analyzed within a memory limit of 8 GB.

The analysis of Java applications is more costly since the applications are larger and more options interact leading to larger constraints. Although we have a median runtime of 16 seconds the runtime for large and complex applications is much higher. For *ProteaJ* we have a runtime of 1.3h and for Adligo even 4.5h. The peak memory usage was 9.6 GB. Unlimited caching of solver calls is the most important factor in memory usage, a different caching strategy could easily be used if memory is an issue.

Currently, LOTRACK's performance is limited to mid-size Java applications. More popular Java applications, such as Hadoop, are much larger. There are three important factors in the performance of LOTRACK. First, the number of edges in the control-flow graph, second, the number of configuration options used and, third, the way configuration options are used. There are multiple patterns in an application that make the analysis more expensive. For example, if objects holding configuration options are passed into every part of the program, LOTRACK has to pass more taints along more edges of the control-flow graph. Uses of configuration values in complicated control-flow structures lead to more complex constraints resulting in longer runtime of the solver. These problems will need to be addressed in future versions of the tool. Still, we can already see how a software design, in which variables holding configuration values have a limited liveness, would make it easier to track configuration options in large applications.

An analysis of the performance showed that the runtime is mostly influenced by calls to the SMT solver. Although each call to the solver is very fast, large applications can lead to a very high number of calls and complex solver queries due to the use of many (symbolic) symbols in imprecise constraints. Additionally, we use the solver to simplify its results so that we can save intermediate results more efficiently, which is computationally expensive as well.

The runtime we currently achieve should still be acceptable for users given the expected usage of LOTRACK. We store LOTRACK's results in a database, the analysis is therefore a one-time cost for each application, i.e., it could be easily run over night, and then manually explored or used for further analyses without additional analysis time.

5.5 Study of Configuration Options in Android Apps and Java applications

To exemplify how our analysis can help researchers and developers understand highly configurable systems, we performed a small empirical study on configurations in 100 Android apps and 10 Java applications.

To study the use of configuration options, we executed our analysis on each program and investigated the configuration map regarding the following research questions:

1) *RQ1: Which Android options are used in practice?* To that end, we observe which configuration options occur in each app's configuration map.
2) *RQ2: How much of the code depends on one or multiple configuration options?* Technically, we use the configuration map to identify which code fragments are mapped to configuration constraints that are not *true*.
3) *RQ3: How frequently do configuration options interact?* Technically, we analyze how many code fragments are mapped to configuration constraints involving more than one option.

9. http://labs.carrotsearch.com/junit-benchmarks.html

TABLE 5: Common Configuration Options [Top 5]

Option	Number of apps using the option
SDK	47
NETWORK	19
MODEL	12
VIBRATOR	12
STORAGE	11

TABLE 6: Interacting Options in Apps [Top 5]

Options	First-order Interactions	Second-order Interactions	Higher-order Interactions (>2)
SDK	10	3	2
WIFI	3	3	2
PHONE	3	1	2
AUDIO	2	2	2
VIBRATOR	5	1	0

4) *RQ4: What is the difference in the use of configuration options in Android apps and Java applications?* We will show how the structure of the software will affect the resulting configuration map.
5) *RQ5: How close is the location of the initial access of a configuration option to its uses in the configuration map?* For each configuration option, we determine the maximum distance from the point of the initial access to the configuration option to the point where the configuration option affects the control-flow. The distance is the smallest, if both points are in the same method, and largest if the points are in different packages.

Regarding RQ1 (*Which Android options are used in practice?*), the most commonly used configuration option is *SDK*, used by 47 apps (Table 5). The option is used to distinguish between the different versions of the Android platform. Depending on the version, different features of the framework can be used. Other commonly used options are *Network* and *Model* options, used by 19 and 12 apps respectively. These options subsume information about availability and state of network component and the model of device.

Regarding RQ2 (*How much of the code depends on one or multiple configuration options?*), the share of statements (in number of Java statements) with constraints ranges from 0% to 72% with a median of 1%. That is, most apps depend on configuration options, but typically only a small amount of their implementation is configuration-specific. Only few outliers contain much configuration-specific code. Large amounts of configuration-specific code are typically due to classes or methods being exclusively used in parts of the code guarded by certain configuration settings. Certain patterns in the code may lead to an initially surprisingly high number of statements with constraints, e.g., an early return based on a configuration option or the use of exception handling.

TABLE 7: Interacting Options in Java applications

Application	None	1st order	2nd order	Higher-order
platypus	64%	29%	7%	0%
kafka-dispatch	34%	48%	15%	0%
Data Consumer	31%	64%	3%	2%
AndSync	100%	0%	0%	0%
ProteaJ	3%	88%	10%	0%
adligo	89%	3%	7%	0%
RemoteREngine	100%	0%	0%	0%
M-Grid	77%	2%	13%	8%
jmxetric	98%	2%	0%	0%
WarGame...	12%	83%	2%	4%
Arithmetic mean	61%	32%	6%	1%

```
01 boolean help = Option.HELP;
02 if(help) {
03   showHelp();
04   return;
05 }
06 [...]
```

HELP

¬HELP

Fig. 12: Simplified use of the common option *help*.

Regarding RQ3 (*How frequently do configuration options interact?*), we investigate not only options but specific constraints to determine to what degree options interact. Table 6 shows the options which are most often part of interactions in Android. For this, we extracted all combinations of options seen in the constraints of the configuration maps. The table also shows the order of interactions which the options are part of. For example, the option SDK is part of 10 different first-order interactions, i.e., interactions between SDK and a single other option. In three cases, SDK interacts with *two* other options and in two cases with *more than two* other options.

Similarly, in Table 7 we see for each Java application in the evaluation set how many of the entries in the configuration map constitute an interaction of the different orders. The results show large differences between the individual application, e.g., *andsync_server* contains no interaction while for *proteaj* first-order interactions (interactions among two options) are very common (88%). The structure an of application and how they use configuration options has a significant impact on the resulting interactions.

Overall, interactions make up only a small fraction of the configuration maps: about 10% of all constraints in the Android apps are first-order interactions, second-order interactions account for 0.1%, and all higher-order interactions 2%. Due to the different uses of configurations of in the set of Java applications, we see a higher number of interacting configuration option. Still, the distribution within the different order is consistent: after a relatively high share of first-order interactions (32%), we see only few second-order (6%) and higher-order interactions.

Our findings that interactions are relatively rare in practice is consistent with previous results on Java applications [40].

Regarding RQ4 (*What is the difference in the use of configuration options in Android apps and Java applications?*), we can see that the share of statements affected by configuration options is significantly different in Android apps (1%) and Java applications (11%). The structure of an application can have a large influence on the configuration map. For example, many command-line applications have an option *help* to describe their usage and an implementation similar to the one shown in Fig. 12.

While it is obvious that the method `showHelp()` is only used when configuration option HELP is enabled, the early

TABLE 8: Maximum distance from configuration API to its effect on the configuration map

Name	None	Same Method	Other Method	Other Class	Other Package
platypus	2	1	-	-	6
kafka-dispatch	1	2	-	-	2
Data Consumer	3	1	-	-	4
AndSync	6	2	-	-	-
ProteaJ	-	-	-	-	6
adligo	-	-	-	-	5
RemoteREngine	4	-	-	-	1
M-Grid	23	3	2	1	9
jmxetric	4	-	2	1	-
WarGameofThrones	4	1	-	2	1
Sum	47	10	2	4	34

return will lead to a configuration map in which almost every statement in the program has the constraint ¬HELP.

Regarding RQ5 (*How close is the location of the initial access of a configuration option to its uses in the configuration map?*), we compared the location (package, class, method) of any API access of an option in our set of Java applications to the locations of all entries for that option in the configuration map (Table 8). Column *None* indicates the number of configuration options in this application that are accessed in the program but have no affect on the configuration map. For example, an option like *port number* is often passed through the program but is not used to make a control-flow decision. *Same Method* shows the number of options that only affect statements in the same method where the respective option was initially loaded. *Other Method*, *Other Class*, and *Other Package* indicates cases where an option's effect on the configuration map becomes increasingly distant to the point where it was initially loaded.

Even if we assume that finding the effect of a configuration option within a method is easy, we see from our results that the majority of options have effects even on other packages.

5.6 Threats to Validity

Due to technical limitations of our implementation (see Section 4), we only support the use of configuration options through their normal API though other ways are possible, e.g., using reflection. In our evaluation of accuracy, we only used a small sample to show the correctness of our implementation due to significant effort for creating reliable oracles. The results are consistent, however, giving confidence to the accuracy of our approach on real software systems.

For the analysis of Android apps, we only looked for configuration options given by the used framework, whereas more options can be defined by each app. This could increase the number of statements depending on configuration options.

6 Comparison to Program Slicing

A configuration map may seem to be similar to the result of a forward program slice when using the instructions loading a configuration option as slicing criteria. Due to the apparent similarities, in the following, we explain conceptual differences and argue that, in software engineering tasks related to configuration options, a configuration map is more useful than a generic program slice. We will also explain how a configuration map could be derived from slicing algorithms with some nontrivial modification.

Configuration maps and program slices are designed with different goals and highlight different aspects of a program. Both identify parts of the program that are directly or transitively influenced by given parts of the program. A configuration map only includes statements that are exclusively reached under certain configurations, whereas a program slice will include all statements in which a configuration value or some derived variable is read. That is, a configuration map focuses exclusively on control-flow decisions influenced by configuration options, whereas a traditional slice includes all statements in a program that are control-flow or data-flow dependent on the slicing criterion (or even just all statements that are data-flow dependent for *thin slicing* [49]). As such, slices on a configuration option are typically much larger than entries for the same configuration option in a configuration map.

One can attempt to derive a configuration map from the result of a program slicing algorithm, but this requires nontrivial changes to how traditional slicing algorithms are used:

(E1) To track multiple configuration options and their interactions one would have to separately slice the program for each configuration option. One could then collect for each line of the program the set of configuration options that influence that line, as illustrated for a simple example in Figure 13.

(E2) To focus on control-flow decisions, one could remove all statements that are data-flow dependent, but not control-flow dependent from the slice. Notice that still both control-flow and data-flow dependencies need to be tracked during slicing, but data-flow only dependent statements can be removed in a post-processing step. This would eliminate assignments and other statements using configuration values that are not relevant for the configuration map, for example, Lines 2 and 9 in Figure 13.

With these two extensions one could build a simple form of a configuration map, but would still see two differences:

(D1) The configuration map includes only statements that are *exclusively* reached under certain configurations. In contrast a slice would include all statements (or control-flow decisions) that can ever be influenced by a configu-

ration option. For example, our configuration map does not include a method that is called once guarded by a configuration option and once irrespective of any option, such as Line 12 in Figure 13. Changing the analysis behavior cannot easily be done in a post-processing step but would require invasive modification of the slicing algorithm.

(D2) Value tracking to explain not only which configuration options may affect a control-flow decision, but also which specific values change the decision, is not easily supported by slicing approaches. In Figure 13, we illustrate the increased precision of constraints with value tracking.

To summarize, slicing targets a different problem than a configuration map, but one could use a slicing algorithm as input to approximate a configuration map (E1 and E2). Tailoring a slicing algorithm for a new purpose this way would likely be a contribution in itself. In our work, we designed an algorithm for building the configuration map from scratch though, because it allows us to express the more specific question of *exclusive reachability (D1)* and allows us to add optional and partial *value tracking (D2)*.

To validate our assumptions about the difference between configuration map and traditional program slices, we designed a small study to answer the following research questions:

1) *RQ1: What is the difference between program slices and configuration maps?* We expect traditional program slices (using only extension E1) to be larger than a corresponding configuration map. To validate this assumption, we compare the number of statements included in the program slice to the number of statements with a non-trivial constraint in a configuration map and qualitatively inspect the differences. We will further discuss the difference by looking at the kind of statements that are included in the slice but not in the configuration map (potentially caused by D1).
2) *RQ2: In how many statements can a configuration map provide additional information?* For this, we check how often value tracking (D2) can be used.

6.1 Study Design

In this study, we analyze the same applications with both LOTRACK and a program slicer, which is part of the analysis tool JOANA [17] and builds upon WALA.[10] There are only few tools which support program slicing for Java available and, of those, JOANA has an easy-to-use API and is actively maintained which makes it a good choice for comparison.

For LOTRACK, we provide a list of methods and fields to access configuration options. We use this list and identified statements which access these options, to build the slicing criteria for the slicer. This way, the slicer uses exactly the same sources as LOTRACK. Both tools use Java bytecode as their input but then use different intermediate formats during analysis. The most robust way to compare the tools' results is again bytecode. We saved both the configuration map and the program slice in terms of bytecode index and method name. This allowed us to perform a fair comparison of the result size.

10. http://wala.sourceforge.net

We used JOANA's interprocedural forward slicer (SummarySlicer) with its default settings. The slicer first creates a *system dependence graph* (SDG) of the program. This SDG can be used to create slices for the different configuration options. The slicing criteria is given as a set of nodes in the SDG. We map SDG nodes to Jimple statements as used by LOTRACK according to their method name and the bytecode index.[11] Using this mapping, we build the slicing criteria from the Jimple statements that were identified in LOTRACK as statements accessing configuration options.

The result of the slicing process is a set of nodes that are part of the program slice. Again, we map the resulting nodes to the configuration map using the method name and bytecode index. Nodes that are specific for the SDG, such as nodes for formal and actual parameters, have no bytecode index and will be ignored. This is consistent with our handling of the Jimple-based results in the configuration map where similar elements are ignored.

The comparison is based on the aggregated slices, i.e., we count each instruction that is contained in any slice only once. This allows for a realistic comparison with the configuration map which too contains each instruction only once.

As subject system for the comparison we use the same set of Java applications as in the evaluation (Table 4). Recently, an extension to JOANA to support Android apps, called JoDroid, was presented [31]. This extension is able to read an apk file and construct an SDG based on the Android life-cycle. Unfortunately, for most apps we were unable to successfully create the SDG given the resources available to us. Our problems with scaling the SDG creation to some apps were confirmed by the JoDroid team.

6.2 Results

Table 9 summarizes the results of the mapping between the program slice and configuration map for each subject system. In three cases, we are unable to perform the slicing and are therefore unable to calculate the size factor. For the application ProteaJ we had run LOTRACK in a way that it supports Java 8 features in the application but in this mode we cannot generate the necessary information to map to slices. For the SDG creation, JOANA needs information on the used libraries. For WarGameOfThrones the analysis could not run with the available resources when including all necessary libraries. For these two applications we only report the size of the configuration maps.

Regarding RQ1, we can confirm our premise that program slices (based on extension E1) are larger than configuration maps based on the number of instructions included. This premise holds for every subject systems though the difference varies between factors of 1.01 up to 6.2. A factor of around 1.0 as seen for *Data Consumer*, means that the slice and the configuration map are roughly equal in size. For the application *AndSync* we see a much larger difference where the slices are more than six times larger than the configuration map. Based on the median of the results, program slices are on average 1.8 times larger than configuration maps.

11. A single Jimple statement may represent multiple bytecode instructions in which case we create the mapping using any of the contained indices.

	Program	Configuration Map		Slicing with extension	
		Simple	Value Tracking	E1	E1+E2
01	`void start() {`				
02	`  int v = Options.SDKVersion;`	{}		{SDK}	{}
03	`  log("...");`	{}		{}	{}
04	`  if(v > 8)`	{}		{SDK}	{}
05	`    log("...")`	{SDK}	SDK>8	{SDK}	{SDK}
06	`    boolean gps = Options.GPS;`	{SDK}	SDK>8	{SDK,GPS}	{SDK}
07	`    if(gps)`	{SDK}	SDK>8	{SDK,GPS}	{SDK}
08	`      [...]`	{SDK,GPS}	SDK>8∧GPS	{SDK,GPS}	{SDK,GPS}
09	`  print(v);`	{}		{SDK}	{}
10	`}`				
11	`void log(string m) {`				
12	`  print(m);`	{}		{SDK}	{SDK}
13	`}`				

Fig. 13: Differences between configuration map and program slice. Lines 2 and 6 are used as the slicing criteria.

TABLE 9: Results of Comparing Program Slices and Configuration Maps

Name	#Instructions in Slices	#Instructions in Config Map	Size Factor
platypus	1,384	701	1.97
kafka-dispatch	114	84	1.36
Data Consumer	284	282	1.01
AndSync	397	64	6.2
ProteaJ	-	10,626	-
adligo	990	574	1.72
RemoteREngine	167	96	1.74
M-Grid	3,121	571	5.47
jmxetric	247	64	3.86
WarGameofThrones	-	772	-
Median	341	189	1.86

Filtering data-flow dependencies (extension E2) should reduce this gap. However, the slice will still cover additional statements that are not exclusively dependent on configuration options (difference D1).

Regarding RQ2, we can see a significant influence of value tracking on the configuration map and therefore an improvement in the quality of the result compared to a program slice. Table 10 shows a classification of every constraint in the configuration map from the Java sample. Constraints that only contain imprecise terms are equivalent to a program slice because from such constraints one can learn which option may influence a statement but has no information about the precise configuration values. A first improvement are constraints with a mix of precise and imprecise terms, e.g., $\neg GPS \wedge WIFI_{\alpha}$, in which case one can make very precise assumptions about one option (GPS has to be disabled) but not the other (it is somehow related to option WIFI). Naturally, a constraint which only consists of precise terms is another improvement. Overall, in Table 10 we see that there are many projects in which every constraint is improved at least partially. For *M-Grid*, we see additional information through value tracking for 34% of all constraints while for *RemoteREngine* value-tracking provides no improvement. Value-tracking is not possible if either the type of option is not supported, e.g., string options are always tracked imprecisely, or if options are propagated and used in the program in a way LOTRACK cannot statically track. For example, *M-Grid* uses many string options, e.g., to define names, which prevent value tracking. There are also integer and Boolean options that are compared to runtime values, which forces LOTRACK to fall back on imprecise tracking. *RemoteREngine* uses string options (resource names) for which LOTRACK can never use value-tracking, and integer options (port numbers) which do not influence the control flow.

6.3 Discussion

Program slicing and the creation of configuration maps are related static analyses. Both start at specific points in a program and track the effect of this statement through the program. Compared to program slicing, a configuration map is a more specialized analysis, for which we empirically validated that it produces smaller and more informative results than slices.

Our goal is to create a configuration map with as many precise constraints as possible to show exactly how configuration options affect a program. Our results show that we can create precise constraints in many cases. For the remaining cases there is a seamless fallback to imprecise constraints which results in many *mixed* constraints that consist of precise and imprecise terms. This allows to iteratively improve our tool to increase the ability to perform value-tracking instead of an all-or-nothing approach. Compared to a generic program slice, a configuration map is tailored for software engineering tasks with a focus on variability. The smaller size and richer information makes a configuration

TABLE 10: Analysis of Configuration Maps Regarding Precise and Imprecise Constraints

Name	precise	mixed	imprecise	#constraints	value-tracking effect
platypus	339	735	0	1,074	100%
kafka-dispatch	53	40	31	124	75%
Data Consumer	152	320	0	472	100%
AndSync	2	0	76	78	4%
ProteaJ	17	10,396	0	10,413	100%
adligo	114	542	85	741	89%
RemoteREngine	0	0	156	156	0%
M-Grid	210	113	632	955	34%
jmxetric	0	0	84	84	0%
WarGameofThrones	0	0	772	772	0%
Android Apps	6,919	8,113	29,358	44,390	34%

map easier to use than a program slice. At the same time, our implementation shows a similar effort is necessary to set up and run the analyses.

7 Related Work

7.1 Slicing and Taint Analysis

There is a long tradition of using program slicing to estimate the influence of certain parts of code, especially to support program comprehension [52], [55]. We already compared our approach to the use of program slicing in detail (Section 6). In another approach in the context of configuration options, a combination of thin slicing and bytecode instrumentation has been used to produce a ranking which configuration options may most likely influence a control-flow decision to assist with configuration errors [58]. ConfDoctor combines program slicing and a stack trace analysis for debugging configuration errors [11]. These techniques help to find relations between concrete program behavior and the configuration. LOTRACK, on the other side, connects information about configuration options with the source code.

LOTRACK is based on taint analysis [45], which is commonly used for information flow analyses related to privacy and security concerns [4], [12], [18]. Other uses of taint analysis include *dynamic* taint analyses to monitor multi-threaded programs [15] and fuzzing programs [16], [19].

Technically, our value tracking is roughly similar to data-flow analyses extended with constraint tracking. For example, this kind of analysis was used to build a path-sensitive null-pointer analysis for C but it is unable to handle complex constraints representing interactions [7].

7.2 Static Analysis for Configurable Systems

There are related static analyses that focus specifically on configuration options:

Ouellet et al. [36] pursued a similar goal of tracking the influence of configuration options with a static analysis. However, their approach does not track data-flow dependencies and, thus, cannot identify an indirect access of configuration options.

Reisner et al. [40] used symbolic execution to explore how configuration options interact in the execution of a set of test cases. They track configuration options as symbolic values and found that interactions are relatively rare and restricted to few options at a time. Their analysis is more accurate but also much more expensive (several computation-weeks per system), and limited to specific test executions, whereas we statically analyze all possible executions tracking only configuration options.

Whereas our goal is to create a configuration map, there are other approaches to support a developer of a configurable system.

Ribeiro et al. [41] use data-flow analysis to explain how data flows among code fragments belonging to different configuration options, to support developers with mechanically derived documentation, called emergent interfaces. In contrast to our work, they know a static configuration map (from preprocessor usage) and track potential data-flow of all other variables, whereas our goal is to track load-time configuration options.

Angerer et al. [2] use a system dependence graph extended with presence conditions to model the impact of configuration options. Based on this model, they perform a configuration-aware *change impact analysis* for load-time configuration options. Compared to this, a configuration map is not specific for a certain use case but can be used in different testing and maintenance scenarios.

SPL$^{\text{LIFT}}$ [9] implements an approach to efficiently apply existing static analyses, e.g., taint analysis, to every variant of a software product line. SPL$^{\text{LIFT}}$ assumes an existing compile-time mapping of code to features based on CIDE [21]. The goal of our approach is to take an unmodified system and extract such a mapping, i.e., the configuration map.

There is comprehensive work on the analysis of variability implemented with the C preprocessor and similar tools [23], [27], [51]. Approaches from this area can create something like a configuration map by analyzing the `#ifdef`s controlling pieces of code. These approaches do not consider indirect influences, e.g., a data-flow in the program that is only active for certain configurations. Additionally, these approaches are only applicable to systems where the use of a preprocessor is common, i.e., C/C++ but not the Java and Android system we are interested in.

7.3 Dynamic Analysis for Configurable Systems

A class of research uses *dynamic* analyses to learn about the effect of configuration options on a program's execution. Toman and Grossman [53] use a *dynamic* taint analysis in their tool STACCATO to identify bugs caused by runtime changes to configuration values. Similar to LOTRACK,

STACCATO uses the configuration API as the source in a taint analysis. While we use an external configuration to specify configuration APIs, STACCATO expects developers to modify their program to access configuration options through a shim provided by the tool. Nguyen et al. [34] sample a program with different configurations and compute the covered lines for each run. This mapping is used to identify *interactions*, which are similar to the constraints in our configuration map.

ConfAid [6] uses a dynamic taint analysis on binary-level to support the search for configuration errors. Similar to LOTRACK, ConfAid uses instructions that read a configuration file as *sources* but the approaches differ in that ConfAid focuses on configuration errors and performs a dynamic analysis.

Variability-aware execution extends the idea of sampling and reasons about all configurations in a single execution run [9], [33], [41], [30]. The result is used to learn about feature interactions but could be used to create a variant of a configuration map as well.

Due to the nature of a dynamic analysis, such approaches can only analyze paths that are actually executed, but this enables an analysis of the effect of runtime changes to configuration values which is out of scope of LOTRACK.

7.4 Other analyses related to configuration options

More generally, our goal of finding a configuration map is related to work on configuration debugging and configuration testing. In configuration debugging, runtime faults are explained in terms of the current configuration and a different configuration is suggested to users to work around the problem using various dynamic and static analyses [37]. Configuration testing determines whether configuration options influence a test case's execution to determine the smallest set of configurations that actually needs to be executed [24], [25], [33], [46]. In contrast to configuration debugging and testing, however, we do not reason about runtime behavior beyond the influence of configuration options.

Furthermore, researchers have investigated whether two patches can interact [10], [44]. Similar to our work they track the potential influence of variations (in their case patches, in ours options) to identify whether multiple changes can interact. After detecting potential interactions they typically focus testing efforts on those code fragments.

8 CONCLUSION

We have extended a standard taint analysis to track load-time configuration options within a program. The analysis produces a configuration map explaining for each code fragment under which configuration options it may be executed. This configuration map can be used for a wide array of maintenance tasks, such as understanding the impact and interactions of configuration options. We have implemented the analysis in our tool LOTRACK and demonstrated its use by studying configuration options in Android apps und Java applications. Our evaluation demonstrated a good accuracy (84% recall and 89% precision) as well as a performance good enough for use on real software systems (8 sec for an average app).

We compared configuration maps to program slices and could show that a configuration map provides more focused information and better information on the effect of configuration options on a program than a generic program slice.

9 ACKNOWLEDGEMENTS

Lillack's work was supported by the German Federal Ministry of Education and Research under grant 01IS15009B. Kästner's work has been supported in part by the National Science Foundation (awards 1318808 and 1552944), the Science of Security Lablet (H9823014C0140), and AFRL and DARPA (FA8750-16-2-0042). Bodden's work was supported by the German Research Foundation (DFG) within the projects RUNSECURE and TESTIFY and the CRC CROSSING, by the state of North Rhine-Westphalia within the graduate school NERD, and by the Heinz Nixdorf Foundation.

REFERENCES

[1] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan. Can we refactor conditional compilation into aspects? In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 243–254, New York, 2009. ACM Press.

[2] F. Angerer, A. Grimmer, H. Prähofer, and P. Grünbacher. Configuration-aware change impact analysis. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, Los Alamitos, CA, 2015. IEEE Computer Society.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, Berlin/Heidelberg, 2013.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, New York, 2014. ACM Press.

[5] D. L. Atkins, T. Ball, T. L. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools: A case study of the Version Editor. *IEEE Trans. Softw. Eng. (TSE)*, 28(7):625–637, 2002.

[6] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, pages 237–250, Berkeley, CA, USA, 2010. USENIX Association.

[7] T. Ball and S. K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Proc. Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, New York, 2001. ACM Press.

[8] I. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 281–290, Washington, DC, 2001. IEEE Computer Society.

[9] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPL^{LIFT}: 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.

[10] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Regression tests to expose change interaction errors. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 334–344, New York, 2013. ACM Press.

[11] Z. Dong, A. Andrzejak, and K. Shao. Practical and accurate pinpointing of configuration errors using static analysis. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*, pages 171–180, Los Alamitos, CA, 2015. IEEE Computer Society.

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.

[13] J.-M. Favre. Understanding-in-the-large. In *Proc. Int'l Workshop on Program Comprehension*, pages 29–38, Los Alamitos, CA, 1997. IEEE Computer Society.

[14] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, 18(4):699–745, 2013.

[15] M. Ganai, D. Lee, and A. Gupta. Dtam: Dynamic taint analysis of multi-threaded programs for relevancy. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 46:1–46:11, New York, NY, USA, 2012. ACM.

[16] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society.

[17] J. Graf, M. Hecker, and M. Mohr. Using JOANA for information flow control in Java programs - a practical guide. In *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*, Lecture Notes in Informatics (LNI) 215, pages 123–138. Springer Berlin / Heidelberg, 2013.

[18] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, pages 303–311, Washington, DC, 2005. IEEE Computer Society.

[19] M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 720–725, New York, NY, USA, 2016. ACM.

[20] D. Jin, X. Qu, M. B. Cohen, and B. Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *Comp. Int'l Conf. Software Engineering (ICSE)*, pages 215–224, New York, 2014. ACM Press.

[21] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320, New York, 2008. ACM Press.

[22] C. Kästner, S. Apel, and M. Kuhlemann. A model of refactoring physically and virtually separated features. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 157–166, New York, 2009. ACM Press.

[23] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824, New York, 2011. ACM Press.

[24] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 57–68, New York, 2011. ACM Press.

[25] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim. SPLat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 257–267, New York, 2013. ACM Press.

[26] D. Le, E. Walkingshaw, and M. Erwig. #ifdef confirmed harmful: Promoting understandable software variation. In *Proc. Int'l Symp. Visual Languages and Human-Centric Computing (VLHCC)*, pages 143–150, Los Alamitos, CA, 2011. IEEE Computer Society.

[27] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114, New York, 2010. ACM Press.

[28] M. Lillack, C. Kästner, and E. Bodden. Tracking load-time configuration options. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 445–456, Los Alamitos, CA, 9 2014. IEEE Computer Society.

[29] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proc. Europ. Conf. Computer Systems (EuroSys)*, pages 191–204, New York, 2006. ACM Press.

[30] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. 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)*, New York, NY, 9 2016. ACM Press.

[31] M. Mohr, J. Graf, and M. Hecker. JoDroid: Adding Android support to a static information flow control tool. In *Proc. Working Conf. Programming Languages (ATPS)*, volume 1337 of *CEUR Workshop Proceedings*, pages 140–145. CEUR-WS.org, 2015.

[32] L. D. Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. Int'l Conf. on Tools and algorithms for the construction and analysis of systems (TACAS/ETAPS)*, pages 337–340, Berlin/Heidelberg, 2008. Springer-Verlag.

[33] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 907–918, New York, 6 2014. ACM Press.

[34] T. Nguyen, U. Koc, J. Cheng, J. S. Foster, and A. A. Porter. iGen: Dynamic interaction inference for configurable software. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, 2016. to appear.

[35] OpenSignal. Android fragmentation visualized. opensignal.com/reports/fragmentation-2013, 2013.

[36] M. Ouellet, E. Merlo, N. Sozen, and M. Gagnon. Locating features in dynamically configured avionics software. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1453–1454, Los Alamitos, CA, 2012. IEEE Computer Society.

[37] A. Rabkin and R. Katz. Precomputing possible configuration error diagnoses. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 193–202. IEEE, Los Alamitos, CA, 2011.

[38] A. Rabkin and R. Katz. Static extraction of program configuration options. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 131–140, Los Alamitos, CA, 2011. IEEE Computer Society.

[39] S. Rasthofer. Personal communication.

[40] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 445–454, New York, 2010. ACM Press.

[41] M. Ribeiro, P. Borba, and C. Kästner. Feature maintenance with emergent interfaces. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 989–1000, New York, 2014. ACM Press.

[42] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[43] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Code generation to support static and dynamic composition of software product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 3–12, New York, 2008. ACM Press.

[44] R. Santelices, M. J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. In *Proc. Int'l Conf. Software Testing, Verification, and Validation (ICST)*, pages 429–438, Los Alamitos, CA, 2010. IEEE Computer Society.

[45] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proc. USENIX Security Symposium (USENIX-SS)*, Berkeley, CA, USA, 2001. USENIX Association.

[46] J. Shi, M. Cohen, and M. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering*, volume 7212 of *Lecture Notes in Computer Science*, pages 270–284, Berlin/Heidelberg, 2012. Springer-Verlag.

[47] J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 33–42, New York, 2010. ACM Press.

[48] H. Spencer and G. Collyer. #ifdef considered harmful or portability experience with C news. In *Proc. USENIX Conf.*, pages 185–198, Berkeley, CA, 1992. USENIX Association.

[49] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 112–122, New York, 2007. ACM.

[50] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Static analysis of variability in system software: The 90,000 #ifdefs issue. In *Proc. USENIX Conf.*, pages 421–432. USENIX Association, 2014.

[51] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proc. Europ. Conf. Computer Systems (EuroSys)*, pages 47–60, New York, 2011. ACM Press.

[52] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[53] J. Toman and D. Grossman. Staccato: A bug finder for dynamic configuration updates. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Dagstuhl Publishing, 2016.

[54] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 125–135. IBM Press, 1999.

[55] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng. (TSE)*, 10(4):352–357, 1984.

[56] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 307–319, New York, 2015. ACM Press.

[57] X. Z. Yunhui Zheng and V. Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 114–235, New York, 2013. ACM Press.

[58] S. Zhang and M. D. Ernst. Which configuration option should I change? In *Proc. Int'l Conf. Software Engineering (ICSE)*, New York, 2014. ACM Press.

Max Lillack is a PhD student at the University of Leipzig, Germany, where he also received his master's degree in 2012. He is interested in re-engineering, specifically in the context of variability.

Christian Kästner is an assistant professor in the School of Computer Science at Carnegie Mellon University. He received his PhD in 2010 from the University of Magdeburg, Germany, for his work on virtual separation of concerns. For his dissertation he received the prestigious GI Dissertation Award. His research interests include correctness and understanding of systems with variability, including work on implementation mechanisms, tools, variability-aware analysis, type systems, feature interactions, empirical evaluations, and refactoring.

Eric Bodden is a full professor for Secure Software Engineering at the Heinz Nixdorf Institute of Paderborn University, Germany. He is further the director for Software Engineering at the Fraunhofer Institute for Engineering Mechatronic Systems. Prof. Bodden has been recognized several times for his research on program analysis and software security, most notably with the German IT-Security Price and the Heinz Maier-Leibnitz Price of the German Research Foundation, as well as with several distinguished paper and distinguished reviewer awards.