Tracking Load-time Configuration Options

Max Lillack
University of Leipzig, Germany

Christian Kästner
Carnegie Mellon University, USA

Eric Bodden
Fraunhofer SIT &
TU Darmstadt, Germany

ABSTRACT

Highly-configurable software systems are pervasive, although configuration options and their interactions raise complexity of the program and increase maintenance effort. Especially load-time configuration options, such as parameters from command-line options or configuration files, are used with standard programming constructs such as variables and if statements intermixed with the program's implementation; manually tracking configuration options from the time they are loaded to the point where they may influence control-flow decisions is tedious and error prone. We design and implement LOTRACK, an extended static taint analysis to automatically track configuration options. LOTRACK derives a configuration map that explains for each code fragment under which configurations it may be executed. An evaluation on Android applications 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.

Categories and Subject Descriptors

K.6.3 [**Software Management**]: Software maintenance

Keywords

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. 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 particularly common and problematic 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. The implementation of configuration options with plugins or conditional compilation provide some static traceability of an option's implementation which is missing in 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 moment 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 [11], 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 [32], 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 naïve forward-slicing algorithm can identify all code potentially affected by a configuration option, directly or indirectly, in practice, it will frequently return slices that

ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2643001.

[1] `https://github.com/MaxLillack/Lotrack`

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, only few configuration options are usually used in any given part of a program. Technically, LOTRACK extends a context, flow, object and field-sensitive *taint analysis* to build a *configuration map* describing how code fragments depend on configuration options.

This paper evaluates LOTRACK in the context of Android applications. Android apps are interesting subjects for studying configuration options, since the Android platform has a reputation for being diverse and fragmented with many different platform versions and hardware features [18]. 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. In a large set of Android apps, we track how configuration options are used and how much code is devoted to implement optional functionality. We find that most 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.

In summary, this paper presents the following original contributions:

- an encoding of the problem of tracking configuration options as a taint-analysis problem,
- a description of how to make use of common characteristics of configuration values in programs to increase the precision of the analysis,
- an implementation based on FlowDroid [3] able to handle Java/Android source code and bytecode, and
- 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.

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 a specific configuration constraint is satisfied. 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 in which configurations the code fragment may be executed in the program, that is, which configuration options or combinations of options need to be selected or deselected. We describe the configuration constraint as a propositional formula over (atomic) 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 specific SDK version is selected. In Figure 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. It is only an excerpt; the whole app uses the shown field six times and in four different classes. This shows the scattered nature of the configuration option's implementation.

```
          class ProxyService {
            static boolean NATIVE_PROXY_SUPPORTED
                                 = Build.VERSION.SDK_INT >= 12;
            public void onSharedPreferenceChanged() {
              String ketHost;
              if (!NATIVE_PROXY_SUPPORTED) {
SDK<12          ketHost = getString(R.string.pref_proxyhost);
                ...
              }
              String command = path + " -host ";
              String result = RootTools.sendShell(command + ketHost);
              ...
            }
          }
          class ConfigurationActivity {
            public void onHelp(View view) {
              Intent intent;
              if (ProxyService.NATIVE_PROXY_SUPPORTED)
SDK>=12         intent = new Intent(this, ProxyConf...);
              else
SDK<12          intent = new Intent(Intent.ACTION_VIEW, uri);
              startActivity(intent);
            }
          }
```

Figure 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 [4, 10, 15]. A configuration map simplifies otherwise potentially daunting tasks, such as removing an obsolete option from the code [6], refactoring the scattered implementation of an option into a module [1, 12, 16], changing the binding time of a configuration option between compile-time and load-time [24], or determining test-adequacy criteria with configuration coverage [30]. 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 Figure 1, having the configuration map highlights the scattered implementation fragments implementing the option's functionality and supports quick navigation.

There are many different strategies to implement configuration options [2], 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 implementa-

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

tion in those plug-ins. Similarly, using conditional compilation, for example using the C preprocessor's #ifdef directives, despite all criticism [9, 28], enables a simple static localization of all scattered code fragments implementing an option with a simple search over those directives [2, 6, 27]. [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. In Figure 1, we show an example of the result of our approach: The Android app loads a configuration option regarding the SDK version, assigns it to a field, and uses it in other locations in the implementation to decide whether to execute additional code. The resulting configuration map identifies that the additional code can only be executed in specific configurations (shown in gray boxes). Note that, in contrast to slicing [32], our configuration map does not include statements that compute with configuration values or values influenced by them (e.g., Line 22 in our example), but only code blocks included or excluded by configuration-related control-flow decisions.

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 Figure 1); the possible values of configuration options are finite and known. How these options are identified is outside the scope of this paper. Possible strategies include manual identification by reading source code and documentation or using existing heuristics and static analysis tools [21].
- 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.

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), our approach relies on standard static-analysis techniques, conservatively abstracting over concrete values, similar to, e.g., program slicing [32]. 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 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 sine of a regular input, but rarely of a configuration option.

The context, flow, object and field-sensitive taint analysis underlying LOTRACK already allows a precise tracking of the use of configuration options.

3. APPROACH

The general 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, e.g. to detect information leaks. To this end, 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 of a tainted value; when a tainted value occurs in a control-flow decision, one knows that all dependent code may depend on this configuration option. To reduce over-approximation and produce an accurate configuration map, LOTRACK additionally tracks specific values as conditional taints for selected configuration options, as we will explain in Section 3.2.

3.1 Taint Analysis for Configuration Options

Conceptually, LOTRACK performs a taint analysis for each configuration option. The analysis taints all values read from configuration options, as identified by a list of the fully qualified names of methods and fields used to access the configuration API. The taint analysis then propagates the 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; for example, if the right-hand side of assignments contains tainted values, their left-hand side is tainted and is considered to be *derived* from the configuration option. When a tainted value is read in the condition of an *if* statement, one knows that the subsequent computations depend on the configuration option. For example, reading from `SDK_INT` in Figure 1 causes the creation of a taint for the field `NATIVE_PROXY_SUPPORTED` in its initialization expression, and due to an indirect information flow also for `intent`; in the two *if* statements, we then see that the control-flow condition depends on a tainted value.

[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.

```
      boolean gpsOn = LocationManager.isProviderEnabled(GPS);
      boolean wifiOn = Settings.WIFI_ON;
      if(!wifiOn)
¬WIFI   log(gpsOn);
```

Figure 2: Example for access and use of configuration options

To create a configuration map, 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* branch depend on the configuration option and thus are associated with it. Similarly, classes and methods exclusively used in paths guarded by tainted conditionals are associated with this configuration option.

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, often leading to massively overapproximated results. In practice, we allow false negatives and only create taints for results of native-method calls or environment interactions if they have been parameterized with a tainted value. For example in our example in Figure 1 value `result` is tainted because it is returned by a call using the tainted parameter `ketHost`. 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 supports such customizations through its configuration.

3.2 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 (Figure 1), we would ideally like to know that Line 7 is only executed if $SDK\_INT < 12$ instead of only knowing that it *somehow* depends on *SDK_INT* (i.e., configuration constraint $SDK\_INT_?$). To that end, we extend the taint analysis to track configuration *values* instead of only configuration options.

To track configuration values, Lotrack implements several extensions to the taint analysis. In particular, Lotrack tracks a taint for each possible value of a variable and tracks a constraint under which configuration this variable has this value. Second, Lotrack does not propagate all taints directly, but analyzes, restricts, and merges constraints at control-flow decisions.

To explain this in more detail, consider a second example in Figure 2. Here Lotrack does not create a single taint for field **gpsOn** explaining that it depends on configuration option *GPS*, but rather two different taints: one with the value $gpsOn_+$ under the condition that *GPS* is selected and one with the value $gpsOn_-$ under the condition that *GPS* is deselected. The same happens for variable **wifiOn** and option *WIFI*. In the control-flow decision **if(!wifiOn)** one knows that the *then* branch will only be executed if option *WIFI* is deselected; which is why the analysis marks the corresponding control-flow edge with the constraint $\neg WIFI$ and propagates all taints along this path only with restricted constraints; in our case it propagates $gpsOn_+$ only under the condition $GPS \wedge \neg WIFI$.

With value tracking, one can directly model constraints on options with small finite domains. The analysis creates a taint for every possible value (e.g., true and false for Booleans and $Version_{1.2}$, $Version_{1.3}$ for versions). For a constraint with an unknown value of configuration option O we use the notation $O_?$. For example, $Version_?$ indicates that the code fragment *somehow* depends on the *version* configuration option, but allows no statement about the concrete version number.

The *taint information* denoted by the variable and the tracked value together with the constraint constitute a *fact*. Generally, facts are simply propagated like taints in a taint analysis. A fact is no longer propagated, however, if the corresponding constraint is unsatisfiable. If a control-flow decision depends on a tainted value, we derive constraints for the control-flow branches by evaluating the branching condition. All facts are propagated along such control-flow branches with the respective *more restrictive* constraint (a conjunction of the fact's previous constraint and the control-flow branch's constraint). If several facts with the same taint information reach the same statement, for instance, at a control-flow merge point, the constraints for these facts are combined as disjunctions, leading to a *less restrictive* constraint. For example, a fact expressing that variable *a* has the value *true* under condition *DEBUG* merged with another fact expressing that the variable has value *true* under condition $\neg DEBUG$ would be merged into a single fact that the variable has always the value *true*.

At the fix point, the analysis has gathered facts with constraints for each reachable statement in the program. To create the configuration map, Lotrack creates a single constraint for each statement. For this, the constraints of all facts at a statement are disjoint.

3.3 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 the tracking of constraints and refer for a more detailed description of the basic taint-tracking mechanisms to the works on FlowDroid [3].

Our value-based taint-tracking algorithm shown in Algorithm 1 requires an inter-procedural control-flow graph and an initial edge as input. To handle the multiple possible entry points to mobile apps, 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 on the level of *summary edges*. An edge consists of a source and target fact, the source and target statement, as well as a constraint. In Line 2 of Algorithm 1 the initial edge with constraint `true`, given by the control-flow graph, is added to a set of edges to be processed. More edges will be created from the algorithm itself as it traverses the control-flow graph.

```
   input  : inter-procedural control-flow graph (icfg),
            initial edge
   output: set of facts
1  Function trackTaints
2      edges ← {initialEdge};
3      result ← {};
4      while edges do
5          edge ← edges.remove;
6          for successor ∈ icfg.successors(edge) do
7              for fact ∈ successor.facts(edge) do
8                  $constr_{cur.} \leftarrow edge.constr$;
9                  $constr_{fact} \leftarrow$ createConstraint(edge);
10                 $constr_{new} \leftarrow constr_{cur.} \wedge constr_{fact}$;
11                 if result contains fact then
12                     $fact.constr \leftarrow fact.constr \vee constr_{new}$;
13                 else
14                     result.add(new Fact(successor, $constr_{new}$));
15                 end
16                 if $constr_{new} \neq false$ then
17                     edges.add(new Edge(successor, fact));
18                 end
19             end
20         end
21     end
22     return result;
23 end
```

Algorithm 1: Taint-tracking algorithm

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). At an API accessing statement, the taint analysis creates new taints for each possible configuration value (taint creation is not shown). For each possible taint at the successor, the constraint required for this taint to be propagated is determined using Algorithm 2.

Algorithm 2 computes the constraint for the propagation of a fact along a control-flow edge using the information in the edge as well as other facts for the same statement. An API access or a branching statement can lead to the creation of new constraints, all other cases will return *true* leaving constraints of facts, which are propagated along the edge, unchanged.

At an API access (Lines 25-31) the algorithm creates an initial edge pointing to the fact representing the accessed option. The algorithm checks whether value tracking is enabled for this configuration option. For value tracking, a constraint is created with respect to the values of the configuration option. At this point, the possible values are already encoded with possibly multiple fact (one per value), which are created in the taint analysis (not shown in the algorithms). In case of Boolean variables, the analysis simply differentiates between *true* and *false* which are mapped to constraints for this configuration option, for example *WIFI* and $\neg$ *WIFI* respectively. To track values of other types, the algorithm models the possible values as different configuration options e.g., $version_{1.2}$, $version_{1.3}$, $version_{2.0}$. If value tracking is not possible, it creates the generic constraint $WIFI_?$ (Line 30).

```
   input  : edge (target statement and fact), all facts at
            target, mapping API to options
   output: constraint for edge
24 Function createConstraint
25     if target is API access and fact is for accessed option
       then
26         option ← target.accessedOption();
27         if option can use value tracking then
28             return $OPTION_{fact.value}$;
29         else
30             return $OPTION_?$;
31         end
32     else if target is if statement then
33         result ← null;
34         for fact ∈ target.facts() do
35             if fact matches if condition then
36                 if result = null then
37                     result ← false;
38                 end
39                 result ← $result \vee fact.cstr$;
40             end
41         end
42         if result ≠ null and is false branch then
43             result ← negate(result);
44         end
45         if result = null then
46             result ← true;
47         end
48         return result;
49     else
50         return true;
51     end
52 end
```

Algorithm 2: Computation of constraint for edge

At a control-flow decision[4] (Lines 32-48), a new constraint is created if the condition of the *if* statement is dependent on a tainted value. The constraints of all *matching* facts are combined as a disjunction to create the most general constraint for this branching statement (Line 39). A fact *matches* a condition, if its variable and value satisfy the condition. For example, the condition in `if(a)` is satisfied only by the variable `a` and value *true.* For operations on Boolean variables, there is at most one matching fact, but for other types of variables multiple matching facts are possible. For a condition like $version \geq 1.3$ and the three possible facts for option *version* shown above, the resulting constraint will be $version_{1.3} \vee version_{2.0}$.

The resulting constraint is used for the branch edge and its negation for the fall-through edge (Line 43). If no fact matched the condition the algorithm returns *true* (Line 46) which indicates that the condition is not dependent on any configuration option and therefore should not change the constraint of any facts.

Line 10 in Algorithm 1 combines the resulting constraint for the edge from Algorithm 2 by conjoining it with the constraint propagated to this edge so far (Line 8). For example,

[4] All programs are analyzed in an intermediate representation where all kinds of control structures are expressed through *if statements* and *gotos*. Expressions are simplified to comparing a variable to another variable or constant.

if a fact has the constraint $\neg A$ and the constraint from the control-flow edge is B, the resulting constraint for the fact will be $\neg A \wedge B$. Facts are not propagated further if the resulting constraint is *unsatisfiable*.

Along different paths, different facts with the same taint information (i.e., the same variable and value) can reach the same statement. In this case, the facts are joined to a single fact disjoining the individual constraints (Line 12).

The fact 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. The algorithm finishes once there are no more edges to process.

In contrast to a regular taint analysis, for value tracking, the order in which the enlisted edges are processed is important. The creation of a new constraint at an *if* statement requires the information about facts reaching this statement. LOTRACK handles this restriction by ensuring that all open edges are processed before continuing with the edges for the *if* statement. A normal statement with a single successor is handled in the order given the control flow. For calls, exit statements and statements with multiple successors (*if* statements), we define *merge-point* statements to describe statement that merge the branches of the control-flow graph. Edges out of these statements will not be processed until all other edges are processed. For calls and exit statements, we define the return sites as merge points. Return sites comprise the statement after the call and possible catch statements from exception handling. For statements with multiple successors, we select the post-dominator of the statement as merge point. This rule ensures, the *then* branch as well as a possible *else* branch is covered before continuing.

3.4 Example

To illustrate the approach, we walk through a nontrivial example shown in Figure 3. On the 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.

Our analysis uses Jimple, which is an intermediate three-address code representation created from Java source code or bytecode. Jimple introduces intermediate variables, partitions complex expressions, and performs other simplifications. In our example, Lines 20 - 26 show a Jimple-like expansion of the Java expression in Line 27. We use this Jimple-like variant in the control flow on the right side to explain the analysis of the statement step-by-step. The intermediate representation simplifies the implementation and explanation of the approach but does not affect the ability to handle the full set of Java.

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`, whose result represents the configuration option A (see Table 1).

Two data-flow facts are created for variable `a` and the possible values (*true* or *false*). For each value, a constraint (A and $\neg A$ respectively) is created based on the configuration option. These constraints are combined with the current constraint *true*, hence, A and $\neg A$ are the final constraints at this point. Figure 3 shows facts as boxes.

Table 1: Configuration API and value tracking information for Example (Figure 3)

API	Option	Value Tracking
`com.company.hasA()`	A	[+,-]
`com.company.hasB()`	B	[+,-]
`com.company.hasC()`	C	[+,-]
`com.company.hasD()`	D	N/A
`com.company.hasE()`	E	[+,-]

At the following if statement (Line 21), Algorithm 2 is used to create for both outgoing edges (branch and fall-through edge) constraints based on the condition (`a = true`) and the two facts at this statement. For the sake of simplicity, the resulting constraints (A for branch edge and $\neg A$ for fall-through edge) are not shown in the figure. The fact with value *true* satisfies the constraint and is propagated along the branch edge to Line 22. The fact for the false case is not propagated along this edge because $\neg A \wedge A$ is not satisfiable.

In Line 22, the call to method `hasB()` is used to access the configuration option B. Two new facts for variable `notB` are created. In this case, the current constraint is A and the newly created facts represent B and $\neg B$. The resulting constraints are $A \wedge \neg B$ and $A \wedge B$. Note how, at this crucial point, the condition value of `notB` is conditionalized further by the truth value of `a`.

At the two possible return points (Lines 24 and 26), the new fact for the variable `z` at the call site is derived from the facts in the called method. At Line 26, for instance, the constraint for `z` is *false* is calculated from the three incoming facts as $\neg A \vee (A \wedge B) \vee (A \wedge B)$, simplified to $\neg A \vee (A \wedge B)$.

Back in method `start` both facts about 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 facts about `z` are not propagated further along.

After the creation of the constraint on option C (Line 7) and its use (Line 8), Line 9 will depend on C. This dependency affects the newly created constraint on option E: $C \wedge E$ for the *true* case and accordingly $C \wedge \neg E$ for *false*. Because `start` returns no value, the analysis will not propagate any facts beyond the return statement at Line 10. The only fact at Line 11 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 value tracking is not used for D. Unlike in the other cases, there is only one fact created with an undefined value and the opaque constraint $D_?$. Each *if* statement with a condition dependent on a imprecisely tracked variable will create the constraint $D_?$ for the branching edge and $\neg D_?$ for the fall-through edge. We can generally not interpret the constraint $\neg D_?$ and only use it to resolve the constraint ($D_? \vee \neg D_? = true$) even though this equation could be incorrect.

While constraints on value-tracked variables become more and more restrictive in case of nested if statements, the constraints on D stay the same. Lines 13 - 15 will both possibly

```
01 public void start() {
02   boolean z = isConfig();
03   if(z)
04     method1();
05   z = false;
06   boolean e = false;
07   boolean c = hasC();
08   if(c)
09     e = hasE();
10     return;
11   int c = hasD();
12   if(c > 1)
13     if(c < 10)
14       return;
15   method2();
16 }
17
18 public boolean isConfig() {
19 // Jimple-like notation:
20 // boolean a = hasA();
21 // if(a) {
22 //   boolean notB = !hasB();
23 //   if(notB)
24 //     return true;
25 // }
26 // return false;
27   return hasA() && !hasB();
28 }
```

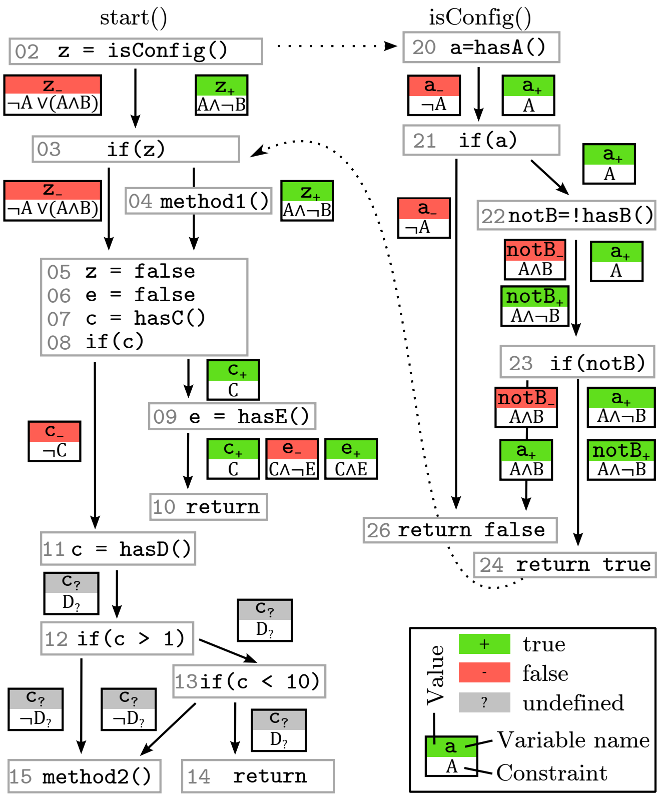

Figure 3: Source code and corresponding call-graph annotated with tracked taints

depend on D but we can make no assumptions on how they relate to some form of the configuration option.

The configuration map is created by disjoining the constraints of all facts at a statement. For example, the configuration map entry for Line 10 is $C \vee (C \wedge \neg E) \vee (C \wedge E) = C$.

4. IMPLEMENTATION

We implemented the approach presented in Section 3 in a tool called LOTRACK. The implementation is based on FlowDroid [3], a tool for taint analysis of Android apps, which in turn is based on Soot [31] as well as SPL$^{\text{LIFT}}$ [7]. Soot is a framework for implementing Java analyses. Input files, i.e. Java source code or Java 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. We use ordered, reduced Binary Decision Diagrams (BDDs) for all operations related to constraints. Because of this, our current implementation is limited to precisely track Boolean options only. Solving inequality constraints on integer values, for instance, would require an appropriate constraint solver.

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 imprecisely. A variability-aware points-to analysis is needed to overcome this limitation. 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 a wide array of maintenance tasks, we have implemented our approach for Java applications and specifically for Android apps. First, we evaluate the *accuracy* of our recovered configuration maps in terms of precision and recall. Second, we evaluate the *performance* of our analysis on a large set of Android apps. Subsequently, we indirectly demonstrate *usefulness* by performing a small empirical study on how configuration options are used within common Android apps and how configuration options interact.

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 vari-

Table 2: Android configuration options (excerpt)

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

ants. Android apps use load-time options to determine the availability of software and hardware functionality at runtime. Configuration options are accessed through standard API, 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 configuration options, we selected 50 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 Boolean options (13 of all 50 configuration options). In Table 2, we show an excerpt of the identified options; a full list is available on the project's web page. Note that the list of configuration options could be easily changed or extended for other systems.

5.1 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 10 Android apps. Subsequently, we automatically extracted a configuration map with LOTRACK and compared it to the manual result, yielding measures of precision and recall.

5.1.1 Oracles

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

As subjects for our evaluation, we randomly selected 10 apps from the FDroid[5] repository of open-source Android apps. The selected apps are shown in Table 3, including some statistics about their size.

To create oracles, we first documented the process that a human developer would take to track configuration options or 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 of the Android framework.

For every subject app, 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 app 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 app. 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 extensive impact, e.g., on called methods.

To evaluate accuracy, we compare the configuration map automatically derived by our tool (from the APK bytecode file) with the manually derived oracle. We count continuous lines of Jimple code as basic *blocks* to prevent a bias towards uses of configuration options that affect a large number of lines. We measure recall as blocks of Jimple code that are correctly mapped to a configuration constraint compared to blocks of Jimple code that are mapped to some configuration constraint in the oracle. We measure precision as blocks of Jimple code that are correctly mapped to a configuration constraint to all blocks of Jimple code that are 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.

5.1.2 Results

In Table 3, we show accuracy of LOTRACK's results: We reach a precision of 85% and a recall of 83%. There was no case of incorrectly detected constraints: the constraints were either correct or missed completely.

In most cases, LOTRACK's result agrees with the oracle. In 19 cases the tool identified constraints for blocks 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 exception handling and methods called only from optional code. We decided to update the oracle with the tool's results in these cases.

LOTRACK missed valid constraint (13 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 11 blocks, where most of the cases seem to be related to overly approximate points-to analysis, a well-known problem that all static analyses share.

[5] https://f-droid.org

Table 3: Comparison of Lotrack's result and manually created oracles on 10 apps

		correct		wrong	
Name	Size (Java LOC)	like oracle	better than oracle	missed	overapproximation
Import Contacts	3,570	3	4	0	0
Nectroid	4,724	4	3	0	2
OSChina	23,280	5	4	3	1
Tinfoil for Facebook	1,364	4	2	2	1
AnySoftKeyboard	18,873	2	1	1	0
Mounts2SD	3,618	2	0	1	0
Impeller	7,389	1	0	0	0
KeePass NFC	546	8	0	1	0
Dolphin Emulator	1,812	1	0	1	0
Document Viewer	50,317	15	5	4	7
sum		45	19	13	11

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.

5.2 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*[6] of five runs after three discarded warm-up runs, on a Core i7 notebook with 3.3Ghz and 16 GB memory. 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.

Of the full sample of 100 apps, we were unable to analyze four apps due to an error reading the Android APK files.

The median time for the analysis is 7.1 seconds; the longest run-time was 181 seconds. The maximum peak memory usage was 2.5 GB.

5.3 Configuration Options in Android Apps

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. We used the same set of subject apps as in our performance evaluation (excluding the same four apps).

To study the use of configuration options, we execute our analysis on each app and investigate the configuration map regarding the following research questions:

1. *RQ1: What 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 in practice?* Technically, we use the configuration map to identify which code fragments are mapped to configuration constraints with one or more configuration options.

3. *RQ3: How frequently do configuration options interact in practice?* Technically, we analyze how many code fragments are mapped to configuration constraints involving more than one option.

Table 4: Common configuration options [Top 5]

Option	Number of apps using the option
`SDK`	38
`NETWORK`	16
`STORAGE`	12
`BLUETOOTH`	7
`AUDIO`	6

Table 5: Distribution of common constraints

Constraint	Share
$SDK_?$	34.4%
$NETWORK_?$	29.1%
$STORAGE_? \wedge SDK_?$	11.0%
$WIFI_?$	5.3%
$LOCALE_?$	4.5%
$LOCATION_?$	3.1%
Other	12.6%

Regarding RQ1, the most commonly used configuration option is *SDK*, used by 38 apps (Table 4). The option is used to distinguish between the versions of Android platform. Depending on the version, different features of the framework can be used. Other commonly used options are *Network* and *Storage* options, used by 16 and 12 apps respectively. These options subsume information about availability and state of network and storage components.

Regarding RQ2, the share of statements (in Jimple notation) with constraints ranges from 0% to 51% with an median of 0.27%. 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 surprising high number of statements with constraints, e.g., an early return based on a configuration option or the use of exception handling.

[6] http://labs.carrotsearch.com/junit-benchmarks.html

Regarding RQ3, we investigate not only options but specific constraints to determine to what degree options interact. Table 5 shows the distribution of common constraints as their share in of all extracted constraints. By far most constraints involve only a single option. By far the most statements depend on option *SDK* without interactions. The most common interaction involves *Storage* and *SDK*. We did not find any interactions among Boolean options that would allow value tracking in our subject systems. Our findings that interactions are relatively rare in practice is consistent with previous results on Java applications [22].

5.4 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 (e.g., using reflection) are possible.

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.

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. RELATED WORK

Our approach can be compared to static program slicing [32], especially *forward slicing* with the API access as the slicing criteria. The slice would contain all program statements affected by a configuration options, including all statements that propagate configuration options. In contrast to slices, our configuration map only includes statements included or excluded due to configuration options and control-flow decisions, but not those statements that read, compute with, or assign configuration-related values. In addition, where possible, we track option values to report also *how* a code fragment depends on a configuration option.

Thin slicing [29] reduces the size of a traditional program slice by considering only *producing* statements, reducing accuracy and producing smaller slices. This technique is optimized towards debugging and program understanding tasks, whereas we select optionally executed statements to support testing and maintenance of configurable software. Recently, a combination of thin slicing and bytecode instrumentation have been used to produce a ranking which configuration options may most likely influence a control-flow decision to assist with configuration errors [33]. This technique helps to find relations between concrete program behavior and the configuration. LOTRACK, on the other side, connects information about configuration options with the source code.

Ouellet et al. [19] 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 indirect access of configuration options.

Reisner et al. [22] 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.

Ribeiro et al. [23] 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.

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 [20]. 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 [13, 14, 17, 26]. 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 [8, 25]. 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.

Technically, our value tracking is roughly similar to data-flow analyses extended with constraint tracking, e.g., used to build a path-sensitive null-pointer analysis for C which is unable to handle complex constraints representing interactions [5]. It is influenced by ideas from variability-aware analysis/execution for product lines where different values can be tracked under different configurations in the same application [7, 17, 23], but tailored to support load-time configuration options with small domains.

7. 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 fragments 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. Our evaluation demonstrated a good accuracy (85% recall and 83% precision) as well as a performance good enough for use on real software systems (7.1 sec on an average app). LOTRACK is not limited to analyzing apps but can be used for Java applications as well. The general concepts should extend to most other imperative programming languages.

8. ACKNOWLEDGEMENTS

Lillack's work is by the BMBF grant 01IS12043B, Kästner's work is supported partly by NSF award 1318808, and Bodden's work by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by the DFG within the project RUNSECURE.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.