Technical Report

Nr. TUD-CS-2012-0239
November 13th, 2012

Transparent and Efficient Reuse of IFDS-based Static Program Analyses for Software Product Lines

TECHNISCHE UNIVERSITÄT DARMSTADT

EC SPRIDE
EUROPEAN CENTER FOR SECURITY AND PRIVACY BY DESIGN

Authors
Eric Bodden (EC SPRIDE / Technische Universität Darmstadt)
Társis Tolêdo (Federal University of Pernambuco)
Márcio Ribeiro (Federal University of Pernambuco)
Claus Brabrand (IT University of Copenhagen)
Paulo Borba (Federal University of Pernambuco)
Mira Mezini (EC SPRIDE / Technische Universität Darmstadt)

Transparent and Efficient Reuse of IFDS-based Static Program Analyses for Software Product Lines

Eric Bodden*, Társis Tolêdo†, Márcio Ribeiro†, Claus Brabrand‡, Paulo Borba† and Mira Mezini§
* Secure Software Engineering Group, EC SPRIDE / Technische Universität Darmstadt, Germany
† Informatics Center, Federal University of Pernambuco, Brasil
‡ Software and Systems Section, IT University of Copenhagen, Denmark
§ Software Technology Group, Technische Universität Darmstadt, Germany

***Abstract*—A software product line (SPL) encodes a potentially large variety of software products as variants of some common code base. Up until now, re-using traditional static analyses for SPLs was virtually intractable, as it required programmers to generate and analyze all products individually.**

In this work, however, we show how an important class of existing inter-procedural static analyses can be transparently lifted to SPLs. Without requiring programmers to change a single line of code, our approach SPL$^{\text{LIFT}}$ automatically converts any analysis formulated for traditional programs within the popular IFDS framework for inter-procedural, finite, distributive, subset problems to an SPL-aware analysis formulated in the IDE framework, a well-known extension to IFDS.

Using a full implementation based on Soot, CIDE and JavaBDD, we show that with SPL$^{\text{LIFT}}$ one can reuse IFDS-based analyses without changing a single line of code. Through experiments using three static analyses applied to four Java-based product lines, we were able to show that our approach produces correct results and outperforms the traditional approach by several orders of magnitude.

I. Introduction

A Software Product Line (SPL) describes a set of software products as variations of a common code base. Variations, so-called features, are typically expressed through compiler directives such as the well-known `#ifdef` from the C pre-processor or other means of conditional compilation. Figure 1a shows a minimal example product line that assigns values through different methods. Figure 1b shows the product obtained by applying to the product line a pre-processor with the configuration $\neg F \wedge G \wedge \neg H$, i.e., a product with feature G enabled and features F and H disabled. Software product lines have become quite popular in certain application domains, for instance for the development of games and other applications for mobile devices. This is due to the tight resource restrictions of those devices: depending on the hardware capabilities of a certain mobile device, it may be advisable or not to include certain features in a software product for that device, or to include a variant of a given feature.

```
void main() {
  int x = secret();
  int y = 0;
  #ifdef F
    x = 0;
  #endif
  #ifdef G
    y = foo(x);
  #endif
  print(y);
}

int foo(int p) {
  #ifdef H
    p = 0;
  #endif
  return p;
}
```

(a) Example SPL

```
void main() {
  int x = secret();
  int y = 0;
  y = foo(x);
  print(y);
}

int foo(int p) {
  return p;
}
```

(b) Product for $\neg F \wedge G \wedge \neg H$

Fig. 1: Example product line: secret is printed if `F` and `H` are disabled but `G` is enabled

Static program analyses are a powerful tool to find bugs in program code [1]–[3] or to conduct static optimizations [4], and it is therefore highly desirable to apply static analyses also to software product lines. With existing approaches, though, it is often prohibitively expensive to reuse existing static analyses. The problem is that traditional static analyses cannot be directly applied to software product lines. Instead they have to be applied to pre-processed programs such as the one from Figure 1b. But for an SPL with n optional, independent features, there are 2^n possible products, which therefore demands thousands of analysis runs even for small product lines. This exponential blowup is particularly annoying because many of those analysis runs will have large overlaps for different feature combinations. It therefore seems quite beneficial to share analysis information wherever possible.

In this work we introduce SPL$^{\text{LIFT}}$, a simple but very effective approach to re-using existing static program analyses without an exponential blowup. SPL$^{\text{LIFT}}$ allows programmers to transparently lift an important class of existing static analyses to software product lines. Our approach is fully inter-procedural. It works for any analysis formulated for traditional programs within Reps, Horwitz and Sagiv's popular IFDS [5] framework for inter-procedural, finite, distributive, subset problems. In the past, IFDS has been used to express a variety of analysis problems such as secure information flow [1], typestate [2], [3], alias sets [6], specification inference [7], and shape analysis [8], [9]. SPL$^{\text{LIFT}}$ automatically converts any such analysis to a feature-sensitive analysis that operates on the entire product line in one single pass. The converted analysis is formulated in the IDE framework [10] for inter-procedural distributed environment problems, an extension to IFDS. In cases in which the original analysis reports that a data-flow fact d may hold at a given statement s, the resulting converted analysis reports a feature constraint under which d may hold at s. As an example, consider again Figure 1. Imagine that we are conducting a taint analysis [1], determining whether information can flow from `secret` to `print`. In the traditional approach we would generate and analyze all $2^3 = 8$ possible products individually, eventually discovering that the product from Figure 1b may indeed leak the secret. SPL$^{\text{LIFT}}$ instead analyzes the product line from Figure 1a in a single pass, informing us that `secret` may leak for the configuration $\neg F \wedge G \wedge \neg H$ (cf. Fig. 1b).

But a reduced analysis time is not the only advantage of a feature-sensitive static analysis. In the area of software product lines, conditional-compilation constructs may add much complexity to the code, and can yield subtle and unusual programming mistakes [11],

[12]. As an example, a plain Java program will not compile if it uses a potentially undefined local variable. In a Java-based software product line, any pre-processor would accept such a program; the programming problem would only manifest later, when the pre-processed program is compiled. When the mistake is discovered, it is laborsome to map the resulting plain-Java error message back to the original product line. Analyzing the product line directly, as in SPL$^{\text{LIFT}}$, circumvents this problem.

To obtain meaningful results, SPL$^{\text{LIFT}}$ further takes feature models into account. A feature model defines a Boolean constraint that describes the set of all feature combinations that a user intends to generate, the SPL's *valid* configurations (or *valid* products). For instance, if we were to evaluate the SPL from Figure 1a under the constraint $F \leftrightarrow G$ (stating that the user intends to generate only products for which both F and G are either enabled or disabled), SPL$^{\text{LIFT}}$ would detect that the secret information cannot leak after all, as it holds that: $(\neg F \wedge G \wedge \neg H) \wedge (F \leftrightarrow G) = \textbf{false}$. Considering a feature model complicates the analysis, which may cause one to expect an increase in analysis cost. Fortunately, our experimental results show that this, in fact, is not usually the case. SPL$^{\text{LIFT}}$ can gain speed by exploiting the model, terminating the analysis for constraint-violating program paths early. This balances out the expected slowdown in many cases.

We have fully implemented SPL$^{\text{LIFT}}$ within a self-written IDE solver on top of the program-analysis framework Soot [13], extending this solver to product lines and connecting Soot to CIDE, the Colored IDE [14], an Eclipse [15] extension for writing and maintaining Java-based software product lines. Using our solution, existing IFDS-based analyses are automatically converted to feature-sensitive versions without changing a single line of code.

We used SPL$^{\text{LIFT}}$ to lift three standard inter-procedural static program analyses, and then applied them to four existing CIDE-based product lines, on which, due to the exponential blowup, the traditional analysis approach takes hours if not years to compute. Our experiments show that SPL$^{\text{LIFT}}$ produces correct results, and outperforms the traditional approach by several orders of magnitude.

At http://bodden.de/spllift/ we make available our full implementation as open source, along with all data and scripts to reproduce our empirical results. To summarize, this paper presents the following original contributions:

- a mechanism for automatically and transparently converting any IFDS-based static program analysis to an IDE-based analysis over software product lines,
- a full open-source implementation for Java, and
- a set of experiments showing that our approach yields correct results and outperforms the traditional approach by several orders of magnitude.

The remainder of this paper is structured as follows. In Section II, we introduce the IFDS and IDE frameworks, along with their strengths and limitations. Section III contains the core of this paper; here we explain the automated lifting of IFDS-based analyses to software product lines. Section IV explains how we take into account the product line's feature model. In Section V we discuss our implementation, while we present our experimental setup and results in Section VI. The work presented in this paper bases itself on previous work presented at the 2012 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security [16]. In Section VII we explain the differences to this paper along with the differences to other related work. Section VIII presents our conclusions.

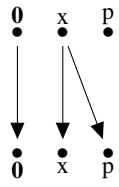
II. The IFDS framework

Our approach builds on top of the so-called *IFDS framework* by Reps, Horwitz and Sagiv [5]. This framework defines a general solution strategy for the inter-procedural, flow-sensitive, and fully context-sensitive analysis of finite distributive subset problems. In this section we present the general concepts behind this framework and illustrate them by an example.

A. Overview of the IFDS Framework

The major idea of the IFDS framework is to reduce any program-analysis problem formulated in this framework to a pure graph-reachability problem. Based on the program's inter-procedural control-flow graph, the IFDS algorithm builds a so-called "exploded super graph", in which a node (s, d) is reachable from a selected start node $(s_0, \mathbf{0})$ if and only if the data-flow fact d holds at statement s. In this setting a "fact" may mean any logical statement that a static analysis can decide, such as "variable x may carry a secret value" or "variables x and y may alias." To achieve this goal, the framework encodes data-flow functions as nodes and edges.

Figure 2 shows how to represent compositions of typical *gen* and *kill* functions as they are often used in static analysis. The nodes at the top represent facts before the given statement s, the nodes at the bottom represent facts after s. The identity function **id** maps each data-flow fact before a statement onto itself. In IFDS, the unique special value **0**, represents a tautology, i.e., a fact that always holds. It is therefore an invariant that two nodes representing **0** at different statements will always be connected. This **0** value is used to generate data-flow facts unconditionally. The flow function α generates the data-flow fact a, and at the same time kills the data-flow fact b (as there is no edge from b to b). Function β, on the other hand kills a, generates b and leaves all other values (such as c) untouched.



non-locally-separable flow function

The unconditional *kill*-and-*gen* functions above can be used to model analysis problems that are *locally separable*, i.e., in which a data-flow fact does not depend on the previous values of other facts. Such problems include the computation of reaching definitions or live variables. Many interesting analysis problems are not locally separable, however, for instance the taint analysis from our example in the introduction. For instance, the function representation to the right could be chosen to model an assignment p=x. Here, x has the same value as before the assignment, modeled by the arrow from x to x, and p obtains x's value, modeled by the arrow from x to p. If p previously held a secret value, then it will only continue doing so if x holds a secret value, too. This is modeled by a missing arrow from p to p.

It is important to note that data-flow facts are by no means limited to simple values such as the local variables in our example. Much more sophisticated abstractions exist, in which facts can, for instance, model aliasing through sets of access paths [1] or even the

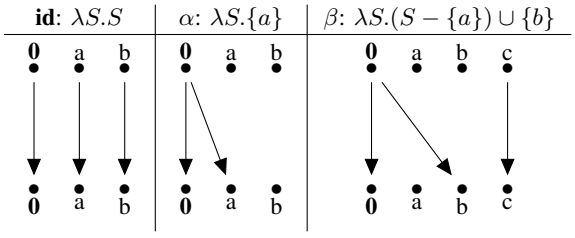


Fig. 2: Function representation in IFDS, reproduced from [5]

abstract typestate of combinations of multiple objects [3]. The IFDS framework itself, however, is oblivious to the concrete abstraction being used; the abstraction is a free parameter to the framework.

B. Different classes of flow functions

Users implement IFDS-based analyses by providing four classes of flow functions that can be denoted as:

- **normal**: modeling flow through a statement that is neither a call nor a return (incl. non-branching, unconditionally branching and conditionally branching statements),
- **call**: modeling flow from a call site to a possible callee,
- **return**: modeling flow from a possible callee back to a return site, i.e., a control-flow successor of the original method-call site, and
- **call-to-return**: modeling intra-procedural flow from just before a call site directly to a given return site.

Normal flow functions track simple intra-procedural flows. Call-to-return functions track intra-procedural flows at call sites; they are useful to propagate, for instance, information about local variables that are *not* passed to a method call, as the call cannot possibly change the value of those variables. Call functions map values from the caller's context to appropriate values in the callee's context. Return functions model the opposite flow, usually from return values to variables that receive those values. Due to this setup, the analysis of each method usually only processes information that is in scope within that method.

C. Example: Taint analysis in IFDS

To encode the assignment of a secret value to a variable `x` we generate a fact x. This fact would be killed to denote that `x` is assigned a definitely non-secret value.

To illustrate this encoding, Figure 3 shows the exploded super graph for our example client of the IFDS framework, the taint analysis applied to the product from the running example (Figure 1b). The analysis starts at the top-left node, which resembles the starting node $(s_0, \mathbf{0})$, s_0 being the starting statement. According to the encoding outlined above, the analysis "taints" the return value at the call to `secret()` by generating the data-flow fact for the variable `x` that holds this value. The analysis then tracks simple assignments to local variables and method parameters as stated above. (For now, let us ignore possible assignments through fields and arrays. We will comment on those in Section V.) At the single call site in the example, the call-to-return function keeps `x` alive, stating that it is still tainted after the call if it was tainted earlier. At the same time, it kills `y` because the call statement is assigning `y` a new value. (Note that at this point `y` is already dead, due to the preceding assignment.) The call function encodes the transfer of actual to formal parameters. Since the only variable in scope within `foo` is `p`, the graph for method `foo` has nodes for `p` but not for `x` nor `y`. In the example, a taint violation is detected by observing the data flow marked with the red/thick arrow: the node for `y` just before the `print` statement is reachable in the graph, which means that a secret value, referred to by `y`, may be printed.

D. The IDE Framework

Sagiv, Reps and Horwitz also developed a more general, more expressive framework than IFDS, called *inter-procedural distributive environment transformers* (IDE) [10]. As in IFDS, the IDE framework models data flow through edges in an exploded super graph. In addition to IFDS, however, IDE allows for the computation of distributive functions along those edges: the label d of each node in the exploded super graph maps to a value v from a second, independent analysis domain, the so-called *value domain* V. The flow functions thus transform environments $\{d \mapsto v\}$ to other environments $\{d' \mapsto v'\}$. Every IFDS problem can be encoded as a special instance of the IDE framework using a binary domain $\{\top, \bot\}$ where $d \mapsto \bot$ states that data-flow fact d holds at the current statement, and $d \mapsto \top$ states that it does not hold [10]. But this binary domain is the least expressive instance of a large set of possible value domains. This we can exploit.

III. Using IDE to lift IFDS-based analyses to Software Product Lines

The main idea of SPL$^{\text{LIFT}}$ is to leverage the gap in expressiveness between IFDS and IDE. To lift IFDS-based data-flow analyses to a feature-sensitive version for software product lines, we replace the binary domain for encoding any IFDS problem as a special instance of the IDE framework by a value domain that consists of feature constraints. The lifting is based on the following principle:

> Assume a statement s that is annotated with a feature constraint F (i.e., `#`**ifdef** `(F)` s `#`**endif**). Then s should have its usual data-flow effect if F holds, and should have no effect if F does not hold.

This principle effectively talks about two different flow functions: one function f^F for the *enabled* case F, and one function $f^{\neg F}$ for the *disabled* case $\neg F$. The idea of SPL$^{\text{LIFT}}$ is to combine both flow functions into one, $f^{\text{LIFT}} := f^F \vee f^{\neg F}$, while using constraints to track traversals of edges labeled with F and $\neg F$ respectively. (The labeling effectively occurs within IDE's value domain V.) The above definition depends on the two notions of a what a statement's "usual effect" should be and what it means to have "no effect". The general principle of disjoining two labeled flow functions f^F and $f^{\neg F}$ into one applies to all flow functions in SPL$^{\text{LIFT}}$. The two component flow functions f^F and $f^{\neg F}$ differ, however, for the different classes of flow functions that we described in Section II-B.

A. Intra-procedural flow functions

First we consider intra-procedural "normal" flow functions, which exist for non-branching, unconditionally branching and conditionally branching statements. Let us first discuss the most simple case of a non-branching statement s. For the *enabled* case F, assume that the statement's original flow function is $f(s)$. In Figure 4a we show on the left the function f^F, which we define as $f(s)^F$, i.e., a copy of $f(s)$ in which all edges are labeled with F, the feature annotation of s. In the figure, we show the edges as dashed because they are conditionalized by this feature annotation. A label F effectively denotes the function $\lambda c.\ c \wedge F$ that will conjoin the incoming constraint c with F when the edge is traversed. In the *disabled* case $\neg F$, statement s is disabled because its feature annotation F contradicts the current product's feature selection. This means that the statement simply does not change the intra-procedural data-flow information at all (cf. the identity function in Figure 2). We hence define $f^{\neg F}$ as shown in the middle column: the identity function labeled with $\neg F$. Once both functions have been labeled, we can combine them into a single flow function as shown in the rightmost column. This function f^{LIFT} effectively represents both previous functions at the same time, as a disjunction $f^F \vee f^{\neg F}$. Edges such as the one from $\mathbf{0}$ to $\mathbf{0}$, which are annotated with F in the one function and with $\neg F$ in the other, are implicitly annotated with **true**. In the following we show such edges as (unconditional) solid black edges. The intra-procedural call-to-return functions are treated exactly the same way.

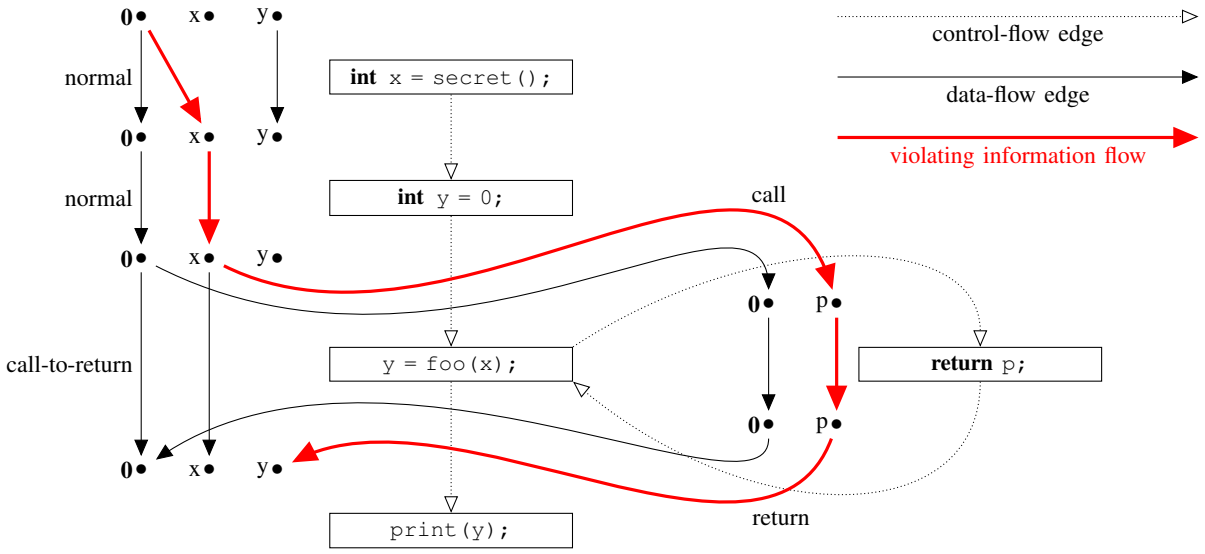


Fig. 3: Exploded super graph for the *single* example product from Figure 1b; main method shown on left-hand side, `foo` shown to the right

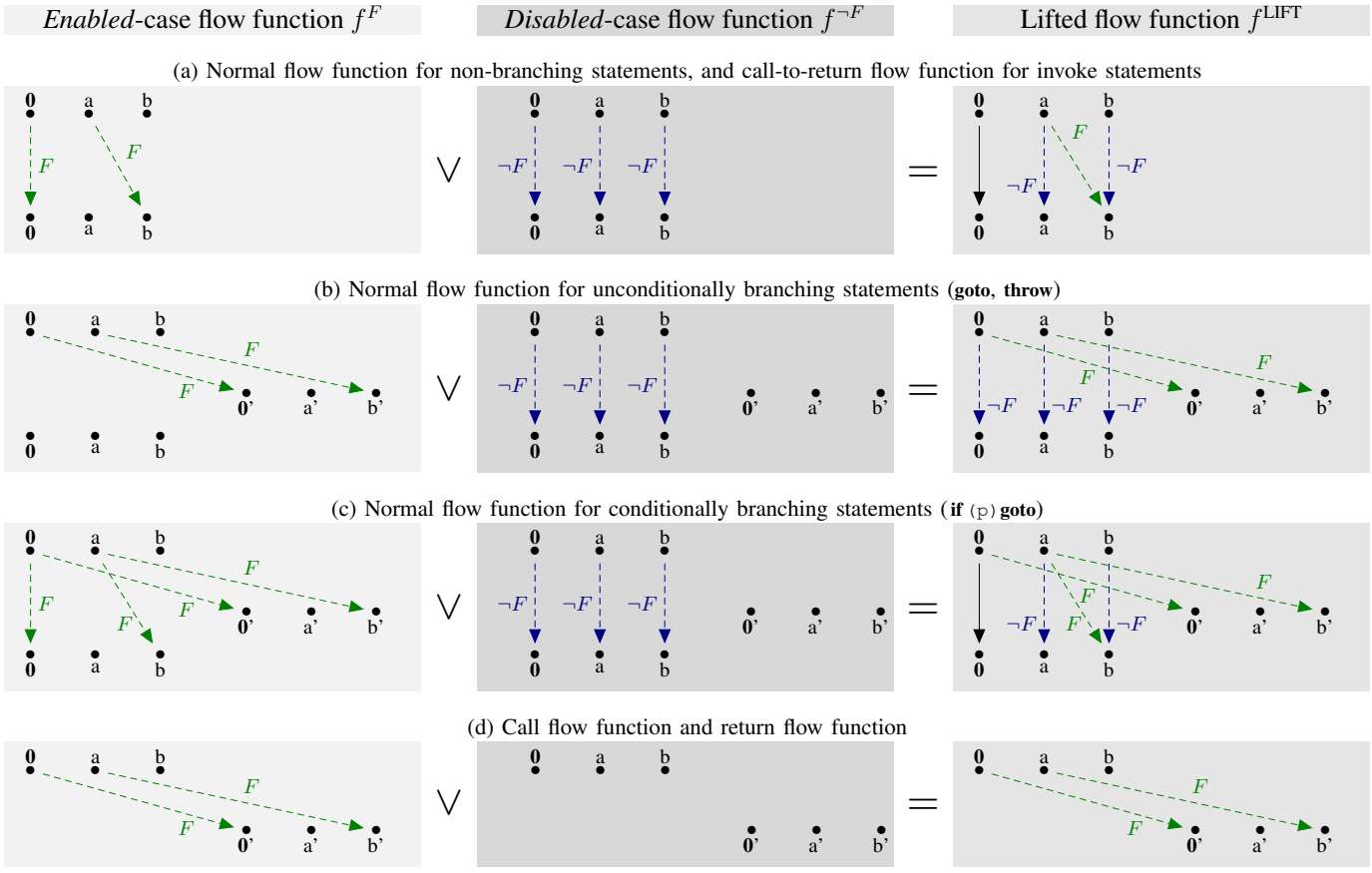


Fig. 4: Lifting of different flow functions in SPL$^{\text{LIFT}}$

But how about intra-procedural flows through branching statements? We conduct our analysis on the Jimple intermediate representation, a three-address code, for which we need to distinguish unconditional branches like **throw** `e` and **goto** `l`, from conditional branches of the form **if** `(p)` **goto** `l`.

Figure 4b shows how we lift flow functions for unconditional branches labeled with a feature annotation F. If the **throw** or **goto** statement is enabled, data flow only exists towards the nodes of the branch target (the primed nodes in Figure 4b). Both edges within this figure are assumed to be labeled with F. If the statement is disabled, data flows only along the fall-through branch, as the branch does not actually execute. Again we use the identity function in this case. Just as before, the lifted flow function f^{LIFT} is obtained through a disjunction of both functions.

For conditional branches of the form **if** `(p)` **goto** `l`, the lifted flow function is basically a combination of the cases for the unconditional branch and the "normal data flow", which models the case in which the branch falls through. We show the respective case in Figure 4c. The *disabled* case $\neg F$ is handled just as for unconditional branches; if a branch is disabled, it just falls through, no matter whether it is conditional or not.

B. Inter-procedural flow functions

The *call* and *return* flow functions model inter-procedural control flows caused by call and return statements. The general idea behind modeling those functions is the same as in the intra-procedural case, however this time we need to consider a different flow function for the *disabled* case, i.e., when $\neg F$ holds. Remember that a call flow function leads from a call site to its callee, and a return flow function from a return statement to just after the call site. Using the identity function to model such situations would be incorrect: If we were to use the identity function then this would propagate information from the call site to the callee (respectively vice versa) although under $\neg F$ the call (or return) never occurs. We must hence use the kill-all function to model this situation, as shown in Figure 4d (middle column). This function kills all data-flow facts, modeling that no flow between call site and callee occurs if the invoke statement is disabled.

C. Computing reachability as a useful side effect

It is an interesting question to ask whether we should conditionalize edges between **0** nodes. As explained earlier, in plain IFDS/IDE, **0** nodes are always connected, unconditionally. We decided for the design shown in Figure 4 where edges between **0** nodes are in fact conditionalized by feature annotations just as any other edges. This has the advantage that, as a side effect, our analysis computes reachability information: a constraint c that SPL$^{\text{LIFT}}$ computes for a node $(s, \mathbf{0})$, i.e., an environment $(s, \{\mathbf{0} \mapsto c\})$, tells the programmer that s is reachable only in product configurations satisfying c.

D. Combining flow functions and initializing flows

As the analysis explores a path in the exploded super graph, we combine all constraint annotations along the path using conjunction. After all, *all* constraints on that path must hold for the flow to be possible. At control-flow merge points we combine constraints using disjunction, as we could have reached the program point along different paths. We initialize the analysis with the constraint **true** at the program's start node $(s_0, \mathbf{0})$, and with **false** at all other nodes.

E. Application to running example

In Figure 5, we show how our generic flow-function conversion rules are applied to our specific running example of our inter-procedural taint analysis, operating on our example product line from Figure 1a. In the figure, the violating information flow leads to the following constraint:

$$(\textbf{true} \wedge \neg F \wedge G \wedge \neg H \wedge G) \vee (\textbf{false} \wedge \neg G) = \neg F \wedge G \wedge \neg H$$

Note that this is exactly the constraint that our introduction promised our analysis would compute.

IV. Considering the SPL's feature model

In the introduction we already hinted at the fact that it is generally necessary to consider a product line's feature model during analysis. Without considering the feature model both the lifted and the traditional feature-oblivious analysis may simply produce useless information as the analysis would be unable to distinguish results for valid configurations from those for invalid ones (cf. Section I).

A. Computing the feature-model constraint from the feature model

Feature models are usually given in the form of a graphical tree representation with additional propositional constraints. As proposed by Batory [17], we translate the model into a single propositional constraint by creating a conjunction of: (i) a bi-implication between every mandatory feature and its parent, (ii) an implication from every optional feature to its parent, (iii) a bi-implication from the parent of every OR group to the disjunction of the components of said group; and (iv) a bi-implication from the parent of every exclusive-OR group, E, to the conjunction of: the pair-wise mutual exclusion of the components E and the disjunction of the components of E.

B. Taking advantage of the feature-model constraint

SPL$^{\text{LIFT}}$ uses the feature model as follows: for an edge label f and a Boolean feature-model constraint m SPL$^{\text{LIFT}}$ implicitly assumes that the edge is instead labeled with $f \wedge m$. Our tool automatically reduces contradictory constraints to **false**, which causes the IDE algorithm to automatically terminate computations for paths that result in such a contradictory constraint.

Due to the fact that the constraint m complicates the overall constraint computation during the buildup of the exploded super graph, one may expect slowdowns over a version of SPL$^{\text{LIFT}}$ that ignores the feature model. However, our particular style of implementation allows for an early termination of the IDE algorithm. As our experiments show, this often counterbalances the slowdown effect. Consider again the example of Figure 5, and in this figure the top-most node labeled with p. During our analysis, when we reach this node, we will have computed the path constraint $\neg F \wedge G$. In combination with the feature model $F \leftrightarrow G$, already at this point we obtain: $\neg F \wedge G \ \wedge \ F \leftrightarrow G \ = \ \textbf{false}$. But this means that any further data flow along this path of the graph is actually infeasible. We can hence terminate further analysis along this path early.

Note that first we tried an even simpler solution: just replace the start value **true** at the program's start node by the feature-model constraint m [16]. While this yields the same analysis results eventually, we found that it wastes performance. The problem is that the IDE algorithm spends most time constructing the graph and internal summary functions, and spends only very little time actually propagating values through the graph. Exchanging the start value only leads to early termination in the propagation phase but not in the costly construction phase. By conjoining the edge constraints with m we terminate early in the construction phase already, saving the algorithm from computing summary functions that, according to the model, would carry contradictory Boolean constraints.

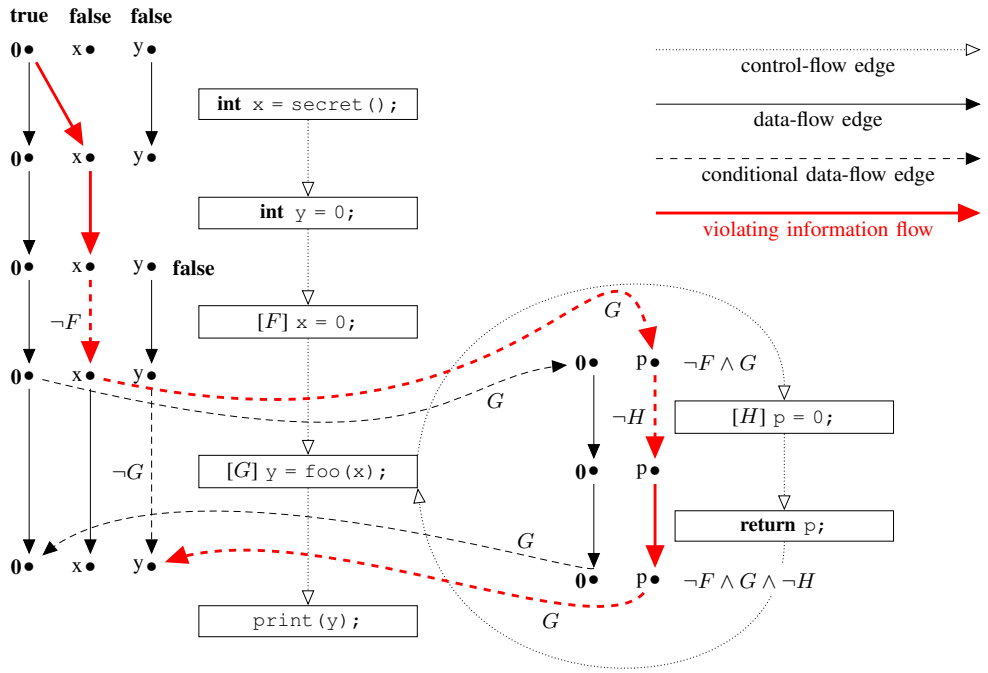


Fig. 5: SPL$^{\text{LIFT}}$ as it is applied to the entire example product line of Figure 1a; an edge labeled with feature constraint C represents the function $\lambda x.\ x \wedge C$. Constraints at nodes represent initial values (at the top) or intermediate results of the constraint computation.

V. Implementation with Soot and CIDE

We have implemented SPL$^{\text{LIFT}}$ based on the Soot program analysis and transformation framework [13], the Colored Integrated Development Environment (CIDE) [14] and the JavaBDD[1] library. We have implemented an IDE solver [10] in Soot that works directly on Soot's intermediate representation "Jimple". Jimple is a three-address code representation of Java programs that is particularly simple to analyze. Jimple statements are never nested, and all control-flow constructs are reduced to simple conditional and unconditional branches. Soot can produce Jimple code from Java source code or bytecode, and compile Jimple back into bytecode or into other intermediate representations.

To be able to actually parse software product lines, we used CIDE, an extension of the Eclipse IDE [15]. In CIDE, software produce lines are expressed as plain Java programs. This makes them comparatively easy to parse: there are no explicit compiler directives such as #**ifdef** that a parser would need to handle. Instead, code variations are expressed by marking code fragments with different colors. Each color is associated with a feature name. In result, every CIDE SPL is also a valid Java program. CIDE forbids so-called "undisciplined" annotations, i.e., enforces that users mark code regions that span entire statements, members or classes. Previous research has shown that undisciplined annotations do exist in real-world projects but that they can often be eliminated by code expansion [18]. Figure 6 shows our running example with the appropriately marked features in CIDE.

One performance critical aspect of our implementation is what data structures we use to implement the feature constraints that SPL$^{\text{LIFT}}$ tracks. After some initial experiments with a hand-written data structure representing constraints in Disjunctive Normal Form, we switched to an implementation based on Binary Decision Diagrams (BDDs) [19], using JavaBDD. Reduced BDDs have the advantage

[1]JavaBDD website: http://javabdd.sourceforge.net/

```
void main() {
    int x = secret();
    int y = 0;
    x = 0;
    y = foo(x);
    print(y);
}

int foo(int p) {
    p = 0;
    return p;
}
```

Fig. 6: Example program in the Colored IDE (CIDE)

that they are compact and normalized, which allows us to easily detect and exploit contradictions. The size of a BDD can heavily depend on its variable ordering. In our case, because we did not perceive the BDD operations to be a bottleneck, we just pick one ordering and leave the search for an optimal ordering to future work.

Current Limitations

Our current implementation is not as sensitive to feature annotations as it could be. This is mostly due to the fact that IFDS/IDE requires the presence of a call graph, and currently we compute this call graph without taking feature annotations into account. While we follow call-graph edges in a feature sensitive fashion (as defined by our call flow function), the feature-insensitive call graph is also used to compute points-to sets. This precludes us from precisely handling situations such as the following:

```
#ifdef F
  x = new ArrayList();
#else
  x = new LinkedList();
#endif
x.add(o);
```

Here, we would conservatively resolve the `add`-call to both `ArrayList.add` and `LinkedList.add`, irrespective of whether F is enabled. In other words, we would propagate the constraint **true** along the respective call edges. This is sound but not maximally precise. Ideally, our points-to analysis should be feature sensitive as well, propagating F to `ArrayList.add` and $\neg F$ to `LinkedList.add`. One could solve this problem by ignoring the pre-computed points-to analysis and handling pointers as part of the IFDS abstraction, e.g., as done by Guarnieri et al. [1].

Due to our tool chain, our implementation is currently limited to software product lines expressed with CIDE. Having said that, there is nothing that precludes our approach from being applied to product lines described in different ways. We are, in fact, currently considering how SPL$^{\text{LIFT}}$ could be applied to C, using the TypeChef parser [20]. The C pre-processor features `#ifdef` constructs that are more expressive than the ones in CIDE, and the C language includes more constructs for unstructured control flow than Java.

Last but not least, SPL$^{\text{LIFT}}$ is limited to analyses expressed in the IFDS framework, which requires that the analyses must have flow functions that are distributive over the merge operator (set union). This is the case for many but not all analysis problems. In the future we plan to investigate to what extent the same ideas are transferable to more expressive static analysis frameworks, for instance weighted pushdown systems [21].

VI. Experiments

We used a set of extensive experiments to answer the following research questions about our implementation:

- **RQ1: Correctness** Does SPL$^{\text{LIFT}}$ compute a sound result?
- **RQ2: Efficiency** How much efficiency do we gain over the traditional feature-oblivious approach?
- **RQ3: Feature Model** What is the cost of using the feature model?

A. RQ1: Correctness

To validate the correctness of our SPL$^{\text{LIFT}}$ implementation, we need an oracle that can tell apart correct from incorrect results. As an oracle, we implemented a second, simplistic analysis that is also feature aware but not constraint based. This analysis essentially corresponds to an inter-procedural version of the analysis *A2* from our earlier publication [22], and we therefore call the analysis *A2* in the remainder of this paper. *A2* operates on the feature-annotated control-flow graph just as SPL$^{\text{LIFT}}$, however unlike SPL$^{\text{LIFT}}$ *A2* is configuration-specific, i.e., evaluates the product line only with respect to one concrete configuration $c = \{F_1, \ldots, F_n\}$ at a time. If a statement s is labeled with a feature F then *A2* first checks whether $F \in c$ to determine whether s is enabled. If it is, then *A2* propagates flow to s's standard successors using the standard IFDS flow function defined for s. If $F \notin c$ then *A2* uses the identity function to propagate intra-procedural flows to fall-through successor nodes only. The implementation of *A2* is so simple that we consider it as foolproof. We hence assume this *A2* implementation as correct, and can therefore use it as an oracle to cross-check the results of SPL$^{\text{LIFT}}$. Whenever *A2* computes a fact r for some configuration c, we fetch SPL$^{\text{LIFT}}$'s computed feature constraint C for r (at the same statement), and check that C allows for c. This ensures that SPL$^{\text{LIFT}}$ is not overly restrictive. The other way around, we traverse all of SPL$^{\text{LIFT}}$'s results (r, c) for the given fixed c, and check that the instance of *A2* for c computed each such r as well (at the same statement). This ensures that SPL$^{\text{LIFT}}$ is as precise as *A2*, i.e., does not report any false positives. Our cross-checks initially helped us to find bugs in the implementation of SPL$^{\text{LIFT}}$, but we quickly converged to an implementation that does not violate any cross-checks any more.

B. RQ2: Efficiency

To evaluate the performance of our approach, we chose four different product lines used in earlier work [22]. We apply to each benchmark subject three different static analyses both emulating the traditional approach (details below) and using SPL$^{\text{LIFT}}$. Table I summarizes some core data about these subjects. BerkeleyDB is a feature-enriched database library. GPL is a small product line for desktop applications, while Lampiro and MM08 are product lines for J2ME applications. Because whole-program analyses require entry points to start with, we programmed driver classes for three of the benchmarks. For the J2ME SPLs, those methods call life-cycle methods usually called by the J2ME framework. For BerkeleyDB, our main class calls the main methods of all driver classes that are shipped with the library. The driver classes are available for download along with all other data and tools to reproduce the experiments.

The third column in Table I gives the total number of features as mentioned in the feature model. The fourth column states the number of features mentioned in feature annotations that are actually reachable from our main classes. For Lampiro this number is surprisingly low. Kästner reports that the current version of Lampiro is unusual in the sense that it contains many dead features and other anomalies, which may be the explanation of this effect [14, pages 131–132]. Column five states the number of configurations over those features, i.e., $2^{\text{Features-reachable}}$. The last column states the number of such configurations that are valid w.r.t. the feature model. For instance, for Lampiro the feature model ended up not constraining the 4 combinations of the 2 reachable features. For BerkeleyDB, we do not know the number of valid configurations. This is because we compute this number as a side-effect of running the traditional approach: for each possible configuration we first check whether it is valid and, if it is, next apply the traditional analysis approach to this configuration. But for BerkeleyDB, due to the $55 \cdot 10^{10}$ possible configurations, this process would take years to complete. For each benchmark/analysis combination we define a cutoff time of ten hours, which is why we cannot report the number of valid configurations for BerkeleyDB.

As analysis clients we used three different generic IFDS-based inter-procedural client analyses. *Possible Types* computes the possible types for a value reference in the program. Such information can, for instance, be used for virtual-method-call resolution [4]. We track typing information through method boundaries. Field and array assignments are treated with weak updates in a field-sensitive manner, abstracting from receiver objects through their context-insensitive points-to sets. *Reaching Definitions* is a reaching-definitions analysis that computes variable definitions for their uses. To obtain inter-procedural flows, we implement a variant that tracks definitions through parameter and return-value assignments. *Uninitialized Variables* finds potentially uninitialized variables. This analysis "generates" variables where they are declared and "kills" them at their first assignment. Again we track parameter and return-value assignments. We provide the source code for all three clients (only approx. 550LOC) in our download package. We include the Java 1.7 runtime libraries in our analysis.

To evaluate the efficiency of SPL$^{\text{LIFT}}$ against the actual traditional approach, we would ideally want to use a pre-processor to generate all possible products, and then apply the traditional analysis approach to each resulting product, an approach which in our earlier work [22] we called *A1*. However, we quickly found that this approach is

so intractable that it would even preclude us from finishing our experiments in due time. This is because the traditional approach would need to generate, parse and analyze every single product. A prerequisite for each analysis is a call graph, whose computation can easily take minutes on its own. Given the large number of possible configurations, it would have taken us years to complete the experiments. (Brabrand et al. [22] did not have this problem because their intra-procedural analyses need no call graph and must only consider up to 2^k combinations where k is the maximal number of features within a single method. For our four benchmark subjects k is always ≤ 9.)

We hence decided to compare SPL$^{\text{LIFT}}$ not to *A1* but instead to our implementation of *A2*, which only requires a single parsing step and call-graph computation. *A2* is thus naturally faster than *A1*, and therefore any performance gain we can show with respect to *A2* would be even higher with respect to *A1*. We found that even with using *A2*, some experiments would still take years to complete, though, which is why we nevertheless use a cutoff-time of ten hours.

For our performance measurements we used a Linux machine with kernel version 2.6.26-1 running on a Quad-Core AMD Opteron Processor 8356 at 2.3GhZ and with 40GB RAM. As Java Virtual Machine we used the Hotspot server VM in version 1.7.0_05, configured to run with 32GB maximal heap space. To make our results easier to reproduce, we configured Eclipse to run all analyses without requiring a user interface. Our implementation is single-threaded. JavaBDD was configured to use the BuDDy[2] BDD package, which implements BDD operations efficiently in native code.

Table II shows our performance comparison between SPL$^{\text{LIFT}}$ and *A2*. In those experiments, both approaches make use of the feature model. For convenience, the second column shows again the number of valid configurations. The *A2* analysis needs to be executed that many times, once for each configuration. The third column shows the time it takes Soot to construct a call graph and points-to sets for the respective benchmark. This time is the same for SPL$^{\text{LIFT}}$ and *A2*, as both require a call graph as prerequisite.

As the table shows, SPL$^{\text{LIFT}}$ outperforms *A2* clearly. While SPL$^{\text{LIFT}}$ never takes more than about 12 minutes to terminate, *A2* always takes significantly longer. In five cases (shown in gray), *A2* even took longer than our cut-off time of ten hours. When this was the case we estimated the rough time it would take *A2* to terminate if we had run it to completion. We computed this estimate by taking the average of a run of *A2* with all features enabled and with no features enabled and then multiplying by the number of valid configurations. (For BerkeleyDB we estimated the number of valid configurations by extrapolating the results obtained within 10 hours.) As this estimation has a relatively low confidence, we only report a very coarse prognosis of days or years.

It is worthwhile noting that, while the absolute performance gain certainly differs depending on the client analysis and chosen product line, the gain is always substantial, and in particular the exponential blowup that *A2* suffers from cannot be observed with SPL$^{\text{LIFT}}$.

Qualitative performance analysis: During our experiments, we found a relatively high variance in the analysis times. As we found, this is caused due to non-determinism in the order in which the IDE solution is computed. As a fixed-point algorithm, IDE computes the same result independently of iteration order, but some orders may compute the result faster (computing fewer flow functions) than others. The non-determinism is hard to avoid; it is caused by hash-based data structures within Soot and our solver. We did find, however, that the analysis time taken strongly correlates with the number of flow functions constructed in the exploded super-graph (the correlation coefficient was above 0.99 in all cases). Moreover, in all our benchmark setups, the *A2* analysis for the "full configuration", in which all features are enabled, constructed almost as many edges as SPL$^{\text{LIFT}}$ did on its unique run. While SPL$^{\text{LIFT}}$ deals with a much more complex analysis domain, i.e., performs more work per edge, our experimental results show that this additional cost is rather low.

[2]BuDDy website: http://buddy.sf.net/

		Features		Configurations	
Benchmark	KLOC	total	reachable	reachable	valid
BerkeleyDB	84.0	56	39	$55 \cdot 10^{10}$	unknown
GPL	1.4	29	19	524,288	1,872
Lampiro	45.0	20	2	4	4
MM08	5.7	34	9	512	26

TABLE I: Key information about benchmarks used

C. RQ3: Feature Model

Regarding a product line's feature model is crucial to any analysis approach for software product lines, as otherwise the analysis would be unable to distinguish results for valid configurations from those for invalid ones, yielding analysis information that would be of little use to clients.

Nevertheless it is an interesting question to ask how high is the cost of regarding the feature model. Table III compares the running time of SPL$^{\text{LIFT}}$ with the running time of an instance of SPL$^{\text{LIFT}}$ where the feature model was explicitly ignored. As our results show, there is usually no significant difference in analysis cost. Exceptions are GPL with *Possible Types* and, to a lesser extent, MM08 with all three analyses. In both cases, the cost of regarding the feature model is clearly observable, albeit far from prohibitive. Apel et al. [23] previously observed as well that regarding the feature model does usually not add much cost during analysis.

Interestingly, those results are quite different from the ones we obtained in our previous approach on intra-procedural analysis for software product lines [22]. There we found that regarding the feature model actually *saved* time. Our explanation for this difference in results is that the intra-procedural analyses from our earlier work use a different, bitset-based constraint representation. This representation is likely less efficient than the BDD-based one in SPL$^{\text{LIFT}}$, which causes the baseline to be much higher. With a higher baseline, the inclusion of the feature model can save more time, as the feature model can help to keep analysis configurations small. With SPL$^{\text{LIFT}}$ the baseline is quite low already. To illustrate this, we included in Table III the average duration of all executed *A2* analyses for the respective setup. Since *A2* is so simple, it is hard to imagine a feature-sensitive analysis (which by its nature considers all configurations, not just one as *A2*) that would complete in less time. As the comparison with those values shows, the analysis time of SPL$^{\text{LIFT}}$ is often actually quite close to this gold standard.

VII. Related Work

The work presented in this paper extends an initial effort on applying the IFDS and IDE analysis frameworks to SPLs [16]. In the earlier work, we only considered "normal" flow functions. Here we significantly refine the technique and implementation. In particular, to achieve soundness and improve performance, we now consider feature model constraints. In our earlier work, we were still speculating about what could be an efficient representation for Boolean constraints. In this work we present a solution based on BDDs that is highly efficient. In our eyes, BDDs are crucial to the

Benchmark	Configurations valid	Soot/CG	Possible Types SPL$^{\text{LIFT}}$	Possible Types *A2*	Reaching Definitions SPL$^{\text{LIFT}}$	Reaching Definitions *A2*	Uninitialized Variables SPL$^{\text{LIFT}}$	Uninitialized Variables *A2*
BerkeleyDB	unknown	7m33s	24s	*years*	12m04s	*years*	10m18s	*years*
GPL	1,872	4m35s	42s	9h03m39s	8m48s	*days*	7m09s	*days*
Lampiro	4	1m52s	4s	13s	42s	3m30s	1m25s	3m09s
MM08	26	2m57s	3s	2m06s	59s	24m29s	2m13s	27m39s

TABLE II: Performance comparison of SPL$^{\text{LIFT}}$ over *A2*; values in gray show coarse estimates

Benchmark	Feature Model	Possible Types	Reaching Definitions	Uninitialized Variables
BerkeleyDB	regarded	24s	12m04s	10m18s
	ignored	23s	11m35s	10m47s
	average A2	*21s*	*9m35s*	*7m12s*
GPL	regarded	42s	8m48s	7m09s
	ignored	18s	8m21s	7m29s
	average A2	*17s*	*7m31s*	*6m42s*
Lampiro	regarded	4s	42s	1m25s
	ignored	4s	48s	1m13s
	average A2	*3s*	*42s*	*49s*
MM08	regarded	3s	59s	2m13s
	ignored	2s	45s	1m49s
	average A2	*2s*	*31s*	*1m37s*

TABLE III: Performance impact of feature model on SPL$^{\text{LIFT}}$. Values in gray show the average time of the *A2* analysis. This number can be seen as lower bound for any feature-sensitive analysis.

performance of SPL$^{\text{LIFT}}$; we found that others do not scale nearly as well for the Boolean operations we require. Finally, for the first time we present empirical evidence of the benefits of our approach.

Our work can also be seen as an extension to an approach by Brabrand et al., who present a number of mechanisms to lift intra-procedural data-flow analyses to SPLs by extending the analysis abstraction with feature constraints [22]. Our approach, on the other hand, lifts information and data-flow analyses to SPLs in an *inter*-procedural fashion, using a different analysis framework, and in particular requiring no extension of the analysis abstraction. In SPL$^{\text{LIFT}}$, the implementation of the IFDS flow functions can remain unchanged. To the best of our knowledge SPL$^{\text{LIFT}}$ is the first work that supports such transparent reuse of analyses. Another crucial difference is our efficient encoding of the distributive Boolean operations through BDDs in the IDE framework.

TypeChef [20], [24] is a parser and analysis engine for product lines written in C. It can parse the entire Linux kernel, including macros and `#`**ifdef** constructs (including undisciplined uses), and performs data-flow analysis. Opposed to SPL$^{\text{LIFT}}$, all analyses are intra-procedural, though, and use a customized analysis domain, thus not providing no support for reusing standard analyses in the way we do. In the future we plan to integrate SPL$^{\text{LIFT}}$ with TypeChef.

Thüm et al. survey analysis strategies for SPLs [25], focusing on parsing [20], type checking [23], [26], model checking [27], [28], and verification [29]–[31]. The surveyed work does not include SPL data-flow analysis approaches, but shares with our work the general goal of checking properties of a SPL with reduced redundancy and efficiency. Similar to SPL$^{\text{LIFT}}$, a number of approaches covered by the survey adopt a *family-based* analysis strategy, manipulating only family artifacts such as code assets and feature model. Contrasting, *product-based* strategies, such as the generate-and-analyze approach we use as baseline, manipulate products and therefore might be too expensive for product lines having a large number of products. Product-based strategies, however, might be appealing because they can simply reuse existing analyses, but this is also the case of the specific family-based strategy proposed here.

In the testing context, Kim et al. use conventional inter-procedural data-flow analysis to identify features that are reachable from a given test case [32]. The test case is then only executed with the SPL products that have these features, reducing the number of combinations to test. They are able to use an off-the-shelf analysis because they express part of the variability using conditional statements, not conditional compilation or other feature tagging mechanisms. This is similar to the technique of configuration lifting [29], which converts compile time variability into runtime variability. In this paper we propose a feature-sensitive analysis to obtain more precision. By applying our family-based analysis followed by their product-based testing one could maybe further reduce the effort to test a SPL. Similar benefits might apply for other testing approaches based on conventional analyses [33] or even feature-sensitive model level analyses [34].

The idea of making dataflow analysis sensitive to statements that may or may not be executed is related to path-sensitive dataflow analysis. Such analyses compute different analysis information along different execution paths aiming to improve precision by disregarding spurious information from infeasible paths [35] or to optimize frequently executed paths [36]. Earlier, disabling infeasible dead statements has been exploited to improve the precision of constant propagation [37] by essentially running a dead-code analysis capable of tagging statements as executable or non-executable during constant propagation analysis.

Predicated dataflow analysis [38] introduced the idea of using propositional logic predicates over runtime values to derive so-called optimistic dataflow values guarded by predicates. Such analyses are capable of producing multiple analysis versions and keeping them distinct during analysis. However, their predicates are over dynamic state rather than SPL feature constraints for which everything is statically decidable.

SPL$^{\text{LIFT}}$ can be applied to a number of contexts, but much motivation comes from the concept of emergent interfaces [39]. These interfaces emerge on demand to give support for specific SPL maintenance tasks and thus help developers understand and manage dependencies between features. Such dependencies are generated by feature-sensitive analyses such as the ones discussed here. In particular, the performance improvements we obtain with our approach are very important to make emergent interfaces useful in practice.

VIII. Conclusion

We have presented SPL$^{\text{LIFT}}$, an approach and framework for transparently lifting IFDS-based static analysis to software product lines using the more expressive framework IDE. Using a set of experiments we were able to show that this approach can outperform the traditional feature-oblivious generate-and-analyze approach by several orders of magnitude. In practice, SPL$^{\text{LIFT}}$ successfully avoids the exponential blowup usually associated with product-line analysis.

This success appears to be due to the following reasons. SPL$^{\text{LIFT}}$ piggybacks onto the user-defined IFDS-based analysis a layer of Boolean feature constraints. The functions generating those Boolean constraints are distributive and hence find a very efficient encoding in the IDE framework. Moreover, we encode all Boolean constraints using minimized binary decision diagrams (BDDs). The Boolean operations we require are conjunction, disjunction, negation and "is false". The two latter operations are constant-time on minimized BDDs. Conjunction and disjunction are, on average, efficient on BDDs, too. The JavaBDD engine we use further memoizes the result of all BDD operations, which speeds up repetitive operations to constant time.

In the future we plan to apply SPL$^{\text{LIFT}}$ to C. Further, we will investigate the performance impact of BDD variable orderings, and to what extent a similar lifting approach can be applied also to static-analysis frameworks that are more expressive than IFDS.

Acknowledgements: Much gratitude goes to to Ondřej Lhoták, who provided useful hints on optimizing BDDs. We also wish to thank the developers of CIDE, JavaBDD and Soot for making their tools available to us and for their continuing support. Thanks to Phil Pratt-Szeliga and Marc-Andre Laverdiere-Papineau, who provided help with analyzing J2ME MIDlets. Thanks also to Sven Apel who provided helpful feedback on an earlier version of this paper. This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED and the Brazilian National Institute of Science and Technology for Software Engineering (INES). Finally, we would like to acknowledge the financial support from CNPq, FACEPE and a CAPES PROBRAL project.

References

[1] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable JavaScript," in *Proc. 2011 int. symp. on Software Testing and Analysis*, ser. ISSTA '11, 2011, pp. 177–187.

[2] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, pp. 9:1–9:34, May 2008.

[3] N. A. Naeem and O. Lhotak, "Typestate-like analysis of multiple interacting objects," in *OOPSLA*, 2008, pp. 347–366.

[4] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for Java," in *OOPSLA*, 2000, pp. 264–280.

[5] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proc. 22nd ACM SIGPLAN-SIGACT symp. on Principles of programming languages*, ser. POPL '95, 1995, pp. 49–61.

[6] N. A. Naeem and O. Lhoták, "Efficient alias set analysis using SSA form," in *Proc. 2009 int. symp. on Memory Management*, ser. ISMM '09, 2009, pp. 79–88.

[7] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," *IEEE TSE*, vol. 34, no. 5, pp. 651–666, 2008.

[8] N. Rinetzky, M. Sagiv, and E. Yahav, "Interprocedural shape analysis for cutpoint-free programs," in *Static Analysis*, ser. Lecture Notes in Computer Science, C. Hankin and I. Siveroni, Eds. Springer Berlin / Heidelberg, 2005, vol. 3672, pp. 284–302.

[9] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn, "Scalable shape analysis for systems code," in *CAV*, pp. 385–398.

[10] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," in *TAPSOFT '95*, 1996, pp. 131–170.

[11] C. Kästner and S. Apel, "Virtual separation of concerns - a second chance for preprocessors," *Journal of Object Technology*, vol. 8, no. 6, pp. 59–78, 2009.

[12] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *Proc. 30th int. conf. on Software Engineering (ICSE'08)*, 2008, pp. 311–320.

[13] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co, "Soot - a Java optimization framework," in *Proceedings of CASCON 1999*, 1999, pp. 125–135. [Online]. Available: www.sable.mcgill.ca/publications

[14] C. Kästner, "Virtual separation of concerns," Ph.D. dissertation, Universität Magdeburg, 2010.

[15] "The Eclipse IDE," http://eclipse.org/.

[16] E. Bodden, "Position Paper: Static flow-sensitive & context-sensitive information-flow analysis for software product lines," in *Proc. ACM SIGPLAN 7th Workshop on Programming Languages and Analysis for Security (PLAS 2012)*, Jun. 2012, to appear. [Online]. Available: http://www.bodden.de/pubs/bodden12static.pdf

[17] D. Batory, "Feature models, grammars, and propositional formulas," in *Software Product Lines (SPLC) 2005*, ser. LNCS, vol. 3714, 2005, pp. 7–20.

[18] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of c code," in *Proc. 10th int. conf. on Aspect-oriented software development*, ser. AOSD '11, 2011, pp. 191–202.

[19] R. Drechsler and B. Becker, *Binary decision diagrams: theory and implementation*. Springer, 1998.

[20] 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. 2011 ACM int. conf. on Object oriented programming systems languages and applications*, ser. OOPSLA '11, 2011, pp. 805–824.

[21] T. Reps, S. Schwoon, and S. Jha, "Weighted pushdown systems and their application to interprocedural dataflow analysis," in *Static Analysis*, ser. Lecture Notes in Computer Science, R. Cousot, Ed. Springer Berlin / Heidelberg, 2003, vol. 2694, pp. 1075–1075.

[22] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba, "Intraprocedural dataflow analysis for software product lines," in *Proc. 11th annual int. conf. on Aspect-oriented Software Development*, ser. AOSD '12, 2012, pp. 13–24.

[23] S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer, "Type safety for feature-oriented product lines," *Automated Software Eng.*, vol. 17, no. 3, pp. 251–300, Sep. 2010.

[24] "Typechef analysis engine," http://ckaestne.github.com/TypeChef/.

[25] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake, "Analysis strategies for software product lines," School of Computer Science, University of Magdeburg, Tech. Rep. FIN-004-2012, Apr. 2012. [Online]. Available: http://www.informatik.uni-marburg.de/~kaestner/tr_analysis12.pdf

[26] C. Kästner and S. Apel, "Type-checking software product lines - a formal approach," in *Proc. 2008 23rd IEEE/ACM int. conf. on Automated Software Engineering*, ser. ASE '08, 2008, pp. 258–267.

[27] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *Proc. 32nd ACM/IEEE int. conf. on Software Engineering - Volume 1*, ser. ICSE '10, 2010, pp. 335–344.

[28] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *Proc. 33rd int. conf. on Software Engineering*, ser. ICSE '11, 2011, pp. 321–330.

[29] H. Post and C. Sinz, "Configuration lifting: Verification meets software configuration," in *Proc. 2008 23rd IEEE/ACM int. conf. on Automated Software Engineering*, ser. ASE '08, 2008, pp. 347–350.

[30] C. H. P. Kim, E. Bodden, D. Batory, and S. Khurshid, "Reducing configurations to monitor in a software product line," in *Proc. First int. conf. on Runtime verification*, ser. RV'10, 2010, pp. 285–299.

[31] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Detection of feature interactions using feature-aware verification," in *Proc. 2011*

26th IEEE/ACM int. conf. on Automated Software Engineering, ser. ASE '11, 2011, pp. 372–375.

[32] C. Hwan, P. Kim, D. Batory, and S. Khurshid, "Reducing combinatorics in testing product lines," in *Proc. 10th int. conf. on Aspect-oriented Software Development (AOSD'11)*, 2011, pp. 57–68.

[33] J. Shi, M. Cohen, and M. Dwyer, "Integration testing of software product lines using compositional symbolic execution," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, J. de Lara and A. Zisman, Eds. Springer Berlin / Heidelberg, 2012, vol. 7212, pp. 270–284.

[34] V. Stricker, A. Metzger, and K. Pohl, "Avoiding redundant testing in application engineering," in *Software Product Lines: Going Beyond*, ser. Lecture Notes in Computer Science, J. Bosch and J. Lee, Eds. Springer Berlin / Heidelberg, 2010, vol. 6287, pp. 226–240.

[35] T. Ball and S. K. Rajamani, "Bebop: a path-sensitive interprocedural dataflow engine," in *Proc. 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '01, 2001, pp. 97–103.

[36] G. Ammons and J. R. Larus, "Improving data-flow analysis with path profiles," *SIGPLAN Not.*, vol. 39, no. 4, pp. 568–582, Apr. 2004.

[37] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 2, pp. 181–210, Apr. 1991.

[38] S. Moon, M. W. Hall, and B. R. Murphy, "Predicated array data-flow analysis for run-time parallelization," in *Proc. 12th int. conf. on Supercomputing*, ser. ICS '98, 1998, pp. 204–211.

[39] M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba, "Emergent feature modularization," in *Proc. ACM int. conf. companion on Object oriented programming systems languages and applications companion*, ser. SPLASH '10, 2010, pp. 11–18.