Information Flow Analysis via Path Condition Refinement

Mana Taghdiri, Gregor Snelting, and Carsten Sinz

Karlsruher Institut für Technologie
76128 Karlsruhe, Germany
{mana.taghdiri,gregor.snelting,carsten.sinz}@kit.edu

Abstract. We present a new approach to information flow control (IFC), which exploits counterexample-guided abstraction refinement (CEGAR) technology. The CEGAR process is built on top of our existing IFC analysis in which illegal flows are characterized using program dependence graphs (PDG) and path conditions (as described in [12]). Although path conditions provide an already precise abstraction that can be used to generate witnesses to the illegal flow, they may still cause false alarms. Our CEGAR process recognizes false witnesses by executing them and monitoring their executions, and eliminates them by automatically refining path conditions in an iterative way as needed. The paper sketches the foundations of CEGAR and PDG-based IFC, and describes the approach in detail. An example shows how the approach finds illegal flow, and demonstrates how CEGAR eliminates false alarms.

Key words: Information flow control, CEGAR, program dependence graph, path condition, abstraction refinement, constraint solving

1 Introduction

Information flow control (IFC) is an important technique for discovering security leaks in software. IFC analyzes the program source code and either discovers a potential illegal flow of information, or guarantees that an illegal flow is impossible. While IFC can take advantage of several well-established program analysis techniques, due to undecidability problems, all sound IFC methods may produce false alarms.

Since false alarms can have a devastating effect on practicability of a security analysis [15], new approaches try to better exploit program analysis and verification technology.[1] The goal is to optimize precision while at the same time, maintaining scalability. In particular, theoretical and practical studies in program analysis have shown that flow-sensitivity, context-sensitivity and object-sensitivity greatly improve the precision of an analysis. However, sensitivity and precision are expensive and can easily destroy scalability, thus, limiting the size

[1] The new DFG Priority Program "Reliably Secure Software" (SPP 1496) represents such an approach in Germany.

of the analyzed program. Therefore, the engineer must carefully decide what hurts more: false alarms or limited program size.

One popular choice is the use of type theory. Many IFC methods are based on security type systems [3, 14, 22], some of which are also available as practical implementations (e.g. [2, 18]). Type systems are sound and reasonably efficient. However, they are not always flow sensitive and context sensitive, and may generate false alarms in the presence of exceptions or recursive methods.

A more precise alternative is an IFC method based on program dependence graphs (PDGs) [12,23,25]. PDGs are flow-sensitive, context-sensitive, and object-sensitive, resulting in fewer false alarms; they also need less annotations than type-based tools such as JIF [11]. PDG-based IFC is sound (it is guaranteed to discover all potential leaks [27]), but much more expensive than type systems. In order to reduce the cost, IBM Research developed a simplified PDG-based IFC that ignores implicit flows (by arguing that they are rare in practice); thus, by sacrificing "a little bit" of soundness, they gain precision (dramatically reduced number of false alarms) as well as scalability up to 500MLOC Java [25].

In this paper, we follow an alternative approach: we explore how to improve precision while maintaining soundness. Our starting point is the PDG-based IFC as described in [10, 12] and implemented in the JOANA analysis tool. JOANA can handle full Java bytecode, and scales up to 50kLOC [10, 11].

JOANA's analysis is based on *path conditions*. PDGs give only a binary answer to the information flow problem: either "there is a potential flow" or "it is guaranteed that no flow exists". To provide a more precise feedback, we have introduced *path conditions* [10, 23] as a device to characterize the circumstances under which a flow may occur. Path conditions are built on top of PDG, exploiting the fact that a flow can happen only along PDG paths. They express conditions over program variables that must be satisfied in order to have an (illegal) flow between two program points. The conditions can be solved using a constraint solver; any satisfying solution provides values to program's input variables which act as a *witness* to illegal flow. Path conditions considerably improve the precision of PDG, but may still generate false alarms due to certain conservative abstractions.

In this paper, we describe a novel approach to eliminating false alarms that may occur in path condition-based IFC. We extend PDGs and path conditions by an instantiation of the counterexample guided abstraction refinement (CEGAR) framework [6]. CEGAR has been successfully used to improve the precision of software model checking [1, 13, 19] and data structure analysis [21, 24]. A number of CEGAR-based program analysis tools (such as [1, 19]) has been successfully used in industry. To our knowledge, however, it has never been exploited in software security and IFC before.

We introduce a novel instantiation of CEGAR that can be applied in the context of information flow analysis. Our approach checks for an (illegal) flow between two given program points by iteratively solving and refining path conditions. It starts from the path condition generated by JOANA for a given pair of program points and solves it using an off-the-shelf, complete constraint solver.

Since path conditions are sound, lack of a satisfying solution guarantees that the program contains no flows between the given points. However, if a solution is found, it must be checked for validity to ensure that it is not a false witness. This is done by executing the program on the input values provided by the solution, and monitoring the program state during the execution. A false alarm will be eliminated by using the constraint solver again, to refine the path condition and reveal more details about the program behavior. This solve-validate-refine process continues until either a valid witness is found, or it is shown that the code contains no flows between the given program points.

The refinement process is monotonic, and thus guarantees termination (for terminating programs). Furthermore, the on-demand nature of our refinement guarantees that only as much code will be analyzed as necessary to check flow between two certain program points. However, the path conditions of larger programs may become intractable after several refinements. Therefore, the analysis time is bounded by a user-provided time-out threshold. In a time-out case, the technique outputs the last satisfying solution as a potential witness for flow.

Compared to the expressive, but purely static PDGs and path conditions, the CEGAR-based approach can eliminate false alarms by incorporating constraint solving, automated test runs, and automatic path condition refinement. The result is a considerable improvement in precision. In this paper, we focus on describing the approach and the underlying ideas and will elaborate on the details of the technique using an example. The technical details of the implementation, and experimental results will follow in a future paper.

2 Program Dependence Graphs

It is a difficult task to construct precise PDGs for full C++ or Java. In particular, context-sensitivity and object-sensitivity require complex and quite expensive ($O(n^3)$) algorithms. We will not describe full details of advanced PDG construction here (see [17] for an overview). Instead, we use a simple example in an imperative subset of Java without objects or procedures to demonstrate the essential aspects of PDGs, as needed to describe our CEGAR-based approach.

Consider the small program of Figure 1 (left). The PDG of this program is given in Figure 1 (bottom). Nodes represent program statements, and there are two kinds of arcs: control dependencies (given by dashed arcs) and data dependencies (given by solid arcs). A statement s is *control dependent* on a statement t iff s will be executed only if a certain expression of t has a specific value. For example, the statements in the body of an `if` or a `while` statement will be executed only if the condition expression evaluates to `true`. In Figure 1, there are control dependencies from node 10 to nodes 11 and 13, and from node 11 to node 12. A statement s is *data dependent* on a statement t iff a variable is assigned in statement t and is used in statement s without being re-assigned in between. That is, data dependencies express nonlocal information transport. In Figure 1, `sum` in statement 8 is used in statements 11 and 12, and `sum` in the left-hand side of statement 12 is used in the right-hand side of the same

```
0. int main(String[] argv) {
1.   int[] a = new int[3];
     // a[0] PUBLIC
     // a[1], a[2] PRIVATE
2.   a[0] = System.in.read();
3.   a[1] = System.in.read();
4.   a[2] = System.in.read();
5.   assert(a[0] > 0);
6.   assert(a[1] > 0);
7.   assert(a[2] > 0);
8.   int sum = 0;
9.   int i = 0;
10.  while (i < 3) {
11.    if (sum == 0)
12.      sum = sum+a[i];
13.    i = i+1;
     }
14.  System.out.println(sum);
   }
```

```
int main(String[] argv_1){
a_1[0] = System.in.read();
a_2[1] = System.in.read();
a_3[2] = System.in.read();
int sum_1= 0;
int i_1 = 0;
while [i_2 = Φ(i_1,i_3);sum_2 = Φ(sum_1,sum_4)]
(i_2<3) {
   if (sum_2==0)
      sum_3 = sum_2 + a_3[i_2];
   [sum_4 = Φ(sum_2,sum_3)] i_3 = i_2 + 1;
}
System.out.println(sum_2);
```

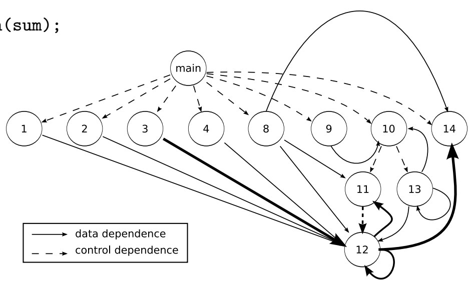

Fig. 1. An example Java fragment, its SSA form, and its PDG.

statement. The latter represents both a so-called loop-carried dependency, and a self-loop.

For simplicity, in this example, asserts are not part of the PDG, and additional control flow due to exceptions is not shown. In general, asserts, exceptions and gotos make the construction of control dependencies much more difficult; the same is true for complex data structures such as arrays, pointers, and objects. Post-dominator trees and a precise points-to analysis are needed to compute dependencies in those cases.

The fundamental property of PDG that makes it relevant for IFC is the *slicing theorem*: If a statement x can influence a statement y in a program P, there must be a path $x \to^* y$ in the PDG of P. If there is no path, it is guaranteed that x cannot influence y. In particular, classical *noninterference* holds if none of the *low* security outputs can be reached via any path from a *high* security variable. Noninterference means that if two initial program states are equivalent on public variables ("low-equivalent"), the final program states after execution must be low-equivalent as well. Noninterference is an effective, if not too strict, security criterion (a discussion of different variants and extensions of noninterference is out of the scope of this paper, see [20] for an overview).

The following theorem states that PDGs provide a sound criterion for noninterference:

$$\Big(\forall x, y \in \mathit{Expression} : x \in \mathit{HIGH}, y \in \mathit{LOW\_Output} \implies x \not\to^* y\Big)$$
$$\implies \Big(\forall s, s' \in \mathit{State} : s \cong_{LOW} s' \Rightarrow [\![P]\!](s) \cong_{LOW} [\![P]\!](s')\Big)$$

Both the intraprocedural and the interprocedural versions of this theorem have been shown correct using machine-checked proofs [26, 27]. It should be noted that in the presence of procedures, the simple concept of a PDG path must be replaced by a so-called context-sensitive backward slice or chop, which guarantees context-sensitivity [12].

Consider the example of Figure 1 again. Suppose that we need to know whether the secret value in `a[1]` can flow to the public output `sum` or not. Since the PDG contains a path (given by thick arcs in Figure 1) from the assignment to `a[1]` in line 3 to the output of `sum` in line 14, it indicates the existence of an illegal flow. This flow, however, is a false alarm. PDGs are precise, but yet, cannot distinguish between different iterations of a loop. In this example, after the first loop iteration, we have `(sum > 0)` and so `a[1]` is never added to `sum`.

2.1 Path Conditions

Path conditions are conditions over program variables, which are necessary for flow along a set of PDG paths. They are computed on top of PDGs to increase the IFC precision. If a path condition $PC(x, y)$ is not satisfiable, it is guaranteed that there is no flow from x to y, even if the PDG contains a path $x \rightarrow^* y$. Path conditions are generated from the control conditions in the statements that govern the execution of a PDG path. The full details are described in [23]; here we only give the essential ideas. As an example, consider the code fragment:

```
1.   a[i+3] = x;
2.   if (i>10 && j<5)
3.      y = a[2*j-42];
```

The PDG of this code (not shown) contains an edge $1 \rightarrow 3$ because – depending on the values of i, j – the array element assigned in line 1 may be used in line 3. The path condition is $PC(1, 3) \equiv (i > 10) \wedge (j < 5) \wedge (i + 3 = 2j - 42) \equiv \mathit{false}$, proving that flow is impossible even though the PDG indicates otherwise. Note that all variables in path conditions are implicitly existentially quantified.

In order to compute path conditions, the program must first be transformed into static single assignment form (SSA) [7]. The SSA form of the example program of Figure 1 (left) is given in Figure 1 (right). Note that assert statements are not part of the SSA form, but contribute to path conditions (see below).

A path condition $PC(x, y)$ is then computed by considering all PDG paths from x to y, and collecting all the conditions that are necessary for each arc in a particular path to be executed. These conditions can result from data flows, control predicates, or the Φ functions in the SSA form. In Figure 1, the path condition for a flow from the initial value of `a[1]` in line 3 to `sum` in line 14 is as follows:

$$\begin{aligned}
\mathrm{PC}\,(3, 14) \equiv\ & (\mathtt{sum}_1 = \mathtt{0}) \wedge (\mathtt{i}_1 = \mathtt{0}) \wedge (\mathtt{i}_2 = \mathtt{1}) \wedge (\mathtt{sum}_2 = \mathtt{sum}_3) \wedge (\mathtt{i}_2 < \mathtt{3}) \wedge (\mathtt{sum}_2 = \mathtt{0}) \\
& \wedge (\mathtt{a[0]} > \mathtt{0}) \wedge (\mathtt{a[1]} > \mathtt{0}) \wedge (\mathtt{a[2]} > \mathtt{0}) \wedge (\mathtt{i}_2 = \mathtt{i}_1 \vee \mathtt{i}_2 = \mathtt{i}_3) \\
& \wedge (\mathtt{sum}_2 = \mathtt{sum}_1 \vee \mathtt{sum}_2 = \mathtt{sum}_4) \wedge (\mathtt{sum}_4 = \mathtt{sum}_2 \vee \mathtt{sum}_4 = \mathtt{sum}_3) \\
\equiv\ & (\mathtt{i}_1 = \mathtt{0}) \wedge (\mathtt{i}_2 = \mathtt{i}_3 = \mathtt{1}) \wedge (\mathtt{sum}_1 = \mathtt{sum}_2 = \mathtt{sum}_3 = \mathtt{sum}_4 = \mathtt{0}) \\
& \wedge (\mathtt{a[0]} > \mathtt{0}) \wedge (\mathtt{a[1]} > \mathtt{0}) \wedge (\mathtt{a[2]} > \mathtt{0})
\end{aligned}$$

The constraints $\mathtt{sum}_1 = \mathtt{0}$ and $\mathtt{i}_1 = \mathtt{0}$ come from the built-in constant propagation (lines 8, 9), $\mathtt{i}_2 = 1$ is attached to the edge $3 \rightarrow 12$ (see [23]), $\mathtt{sum}_2 = \mathtt{sum}_3$ is attached to $12 \rightarrow 14$ (see [23]), the "$\vee$" constraints come from Φ functions, and the other constraints are control conditions of the path or the assert predicates.

This path condition is satisfiable, implying that the secret can allegedly flow to the public output. Hence, although incorporating path conditions into PDG-based IFC reduces many false alarms and makes the analysis considerably more precise, it may still cause false alarms due to conservative abstractions. This example was chosen to demonstrate the dynamic effects (multiple loop iterations) that cannot be captured by static analysis alone, and thus path condition alone cannot eliminate this false alarm. We will describe in the following section how CEGAR refines the path condition, and eliminates the false alarms.

The actual implementation of path condition generation incorporates various optimizations to make the technique scalable to large PDGs. In fact, this IFC analysis is fully implemented in the tool JOANA, and is based on the precise PDG algorithms for concurrent programs as described in [9, 16]. The JOANA PDG can handle full Java bytecode and scales up to 50kLOC. As of today, path conditions can handle an imperative subset of Java and scale to a few kLOC.

3 Path Condition Analysis

The goal of our analysis is to make path condition-based IFC more precise, i.e. if we report that a flow is possible, we will generate a witness whose execution truly illustrates the information flow. To achieve this goal, we eliminate false witnesses fully automatically. However, because analyzing the full code of a program often becomes intractable, we introduce a novel approach in which the precision of the analysis increases incrementally: the technique keeps looking for better witnesses – those that represent the program behavior more closely – in an iterative fashion, ensuring that only as much information about the code is analyzed as actually necessary to establish or refute a flow between two certain program points.

Our analysis is a novel instantiation of the counterexample guided abstraction refinement (CEGAR) framework [6]. It starts with an initial (possibly imprecise) path condition, and follows a fully automatic solve-validate-refine loop until either a definite outcome is reached, or the analysis time exceeds a user-provided threshold. In the former case, an absolutely valid witness is returned, or the program is guaranteed not to contain a flow. In the latter case, the last witness found will be returned as a potential witness. Figure 2 gives an overview of the technique. It consists of the following steps:

- *Generating path conditions.* The technique described in the previous section is used to generate path conditions as an initial approximation of the conditions necessary for an information flow to occur.
- *Checking path conditions.* A constraint solver, namely a SAT Modulo Theory (SMT) solver, is used to check the satisfiability of the path condition. Since the input logic of the SMT solver is decidable, the solver is guaranteed to find a solution when one exists. If the solver cannot find a solution that satisfies

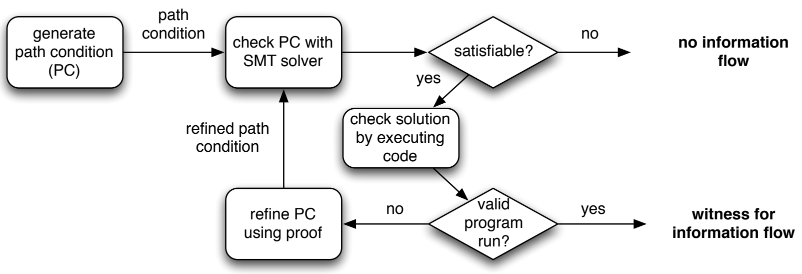

Fig. 2. Path Condition Refinement Loop

the path condition, no flow exists between the two program points, and the analysis terminates. This is because path conditions soundly abstract the necessary conditions for flow.

– *Checking validity of a solution.* If the solver finds a solution, it represents a potential information flow. The solution will be checked for validity to ensure that it is real and not just a spurious alarm. This is done by executing the program code, setting input variables according to the solution provided by the SMT solver, and monitoring the program state as the code runs. If the solution conforms to the actual execution, the analysis terminates, and the solution is returned as an absolutely valid witness to the flow.
– *Refining path conditions.* If the execution of the solution fails, i.e. we arrive at a program state that contradicts the solution, the path condition must be refined to exclude the current solution. To do this, the SMT solver is used again, to compute a proof of unsatisfiability [28] out of the failed execution attempt. The proof encodes those parts of the program that make the current solution spurious. It is conjoined with the previous condition to form a refined path condition which will be solved by the next iteration of our analysis loop.

Each iteration of the algorithm monotonically extends the path condition. The path condition is an abstraction of the code, so in the limit, it is equivalent to the code, and termination is therefore guaranteed for all terminating programs. In practice, however, the path condition can become intractable, and thus, a user-defined run-time threshold is needed for termination.

3.1 Path Condition Checking

A path condition is an expression in a quantifier-free fragment of first-order logic involving equalities and inequalities, arithmetic (linear and non-linear, integer and floating point), arrays, and uninterpreted functions (to model object attributes). Many quantifier-free fragments of first-order logic are decidable. For example, when arithmetic operations are limited to linear arithmetic, or when the data types are represented by fixed-length bit-vectors (e.g. by modeling Java integers as bit vectors of size 32), the formulas become decidable. Combinations of such decidable first-order logics (*theories*) can be solved using SMT (*satisfiability modulo theory*) solvers (see e.g. [4, 5, 8]). Common combinations include QF_AUFLIA (quantifier-free linear integer arithmetic, integer arrays, and uninterpreted functions) and QF_AUFBV (quantifier-free formulas over the theory of bitvectors, bitvector arrays, and uninterpreted functions).

Translation of path conditions to the input language of an SMT solver requires handling scalars, arrays, object attributes, and their operations. For these purposes, a solver for the theory QF_AUFBV is sufficient, as it can handle integers (with a fixed bit-width), arrays, and object-oriented attributes (via uninterpreted functions). Using range-restricted integers is an advantage for our application, as it (i) enables a precise handling of Java integers, including range overflows, and (ii) supports non-linear operations such as multiplication, division, or bit-operations.

Arrays in SMT solvers for QF_AUFBV are modeled as functions mapping bit-vectors (the indices of the array) to bit-vectors (the values of the array)[2]. The array a of Figure 1, for example, is represented as $a : \mathbb{B}^2 \rightarrow \mathbb{B}^{32}$ where $\mathbb{B}^2$ denotes a bit-vector of size 2 that encodes the indices, and $\mathbb{B}^{32}$ denotes a bit-vector of size 32 that encodes the integer values. Object attributes are translated using uninterpreted functions over fixed-sized bit-vectors, following a relational view [24] of the heap.

3.2 Code Execution and Witness Validation

Having found a solution to the path condition, we know that there is a potential information flow. To check whether the solution represents a real flow or not, we only need to check whether it gives a valid execution of the code; the conditions necessary for the flow were already encoded in the initial path condition, and thus hold in any solution. We execute the program, setting input values as indicated by the solution[3]. During the execution, we compare the program state to the variable assignment provided by the solution, looking for discrepancy.

We run the program step by step, like a debugger does, and at each step observe the variables that also occur in the path condition. In order to check for a contradiction with the solver's solution, we have to map program variables to variables present in the path condition. To achieve this, we use a function $\texttt{SSA}(x, \pi)$, which associates a program variable x at program location π with its SSA counterpart $x_i = \texttt{SSA}(x, \pi)$. This function can be created during the SSA transformation of the program.

As the solution from the SMT solver provides concrete values for all variables occurring in the path condition, it can easily be checked against the variables' values of the current program state. Having found a contradiction at some program state, we store that program state, say π_{i_0}, and stop the witness validation

[2] Note that arrays with large index ranges are not a problem, as the computational complexity depends on the number of array accesses, and not the range of the indices.

[3] If an input variable is not in the path condition, it can be set to any arbitrary value.

process. (For statements inside loops we have to check further iterations to detect a contradiction because the flow into the loop need not happen in the first iteration.) If no contradiction occurs and the program terminates, the program run is valid, and thus an information flow really occurs. We can thus report the SMT solver's solution as a witness for this information flow.

3.3 Path Condition Refinement

The refinement phase strengthens the current path condition to guarantee that the current spurious solution (witness) will never be found by the solver again. It conjoins the path condition with additional constraints that encode the reason that this solution does not represent a valid behavior of the program.

In order to compute those additional constraints, we perform a *symbolic execution* of the program. The symbolic execution mimics the actual concrete trace that was executed in the validation phase (see Sec. 3.2), but starts from symbolic inputs rather than concrete ones. During the execution, when a variable (or an object attribute) is updated, its value is computed symbolically. The computed symbolic expressions, however, are valid only for this particular execution trace; the values of variables in other traces are unknown. Therefore, the symbolic execution also keeps track of all branch conditions that are taken during this execution trace. Consider the following code fragment as an example:

```
1.   x = input();
2.   if (x>0)
3.      y = x+5;
4.   z = y + 1;
```

Suppose that the concrete execution trace uses `x=5` as the actual input and thus, the branch condition holds in this trace. The symbolic execution starts by using a symbolic, uninterpreted constant for the input, namely `X0`, and mimics the concrete trace. That is, after line 2, it collects the condition `(X0>0)`, after line 3, it computes `X0+5` for the variable `y`, and after line 4, it computes `X0+6` for `z`. The final symbolic values of `x`, `y` and `z`, denoted by `x'`, `y'` and `z'` respectively, are given by:

$$\texttt{x'} = \texttt{x} = X0 \tag{1}$$

$$\texttt{y'} = \texttt{if } (X0 > 0) \texttt{ then } X0 + 5 \texttt{ else ANY} \tag{2}$$

$$\texttt{z'} = \texttt{if } (X0 > 0) \texttt{ then } X0 + 6 \texttt{ else ANY} \tag{3}$$

We define an *execution point* α_t to refer to a point in an execution trace t. Since a trace can execute the same program point several times – for example, those that are included in a loop body – several execution points can correspond to the same point in the program. We use $exec_t(\pi)$ to denote the set of all execution points of a trace t that correspond to a program point π. Furthermore, we use $\texttt{sym}(v, \alpha_t)$ to represent the symbolic expression computed for a variable v at an execution point α_t, and $\texttt{guard}(\alpha_t)$ to denote the conjunction of all symbolic branch conditions under which the control reaches an execution point α_t of a trace t.

Assuming that the current solution represents a false witness, the concrete execution trace contains an invalid program point π_{i_0} whose executions do not match the variable assignment of the solution. Consequently, the symbolic state at none of the executions of π_{i_0} will match the SSA variables of π_{i_0} either. More precisely, at any execution point α_t that corresponds to π_{i_0}, the symbolic expression $\texttt{sym}(v_i, \alpha_t)$ of some variable v_i will not be equal to $\texttt{SSA}(v_i, \pi_{i_0})$. Thus, the following logical formula will be unsatisfiable with respect to the current solution:

$$\bigvee_{\forall \alpha_t \in exec_t(\pi_{i_0})} \bigwedge_{\forall v_i} \Big(\texttt{SSA}(v_i, \pi_{i_0}) = \texttt{if guard}(\alpha_t) \texttt{ then sym}(v_i, \alpha_t) \texttt{ else ANY} \Big)$$

The use of the guard conditions ensures that the symbolic expressions computed along different execution traces do not contradict each other, and thus the constraints that will be added to refine the path condition are sound. The above formula is then simplified by transforming `if` expressions to logical implications, and normalized so that the top-level operator is a conjunction. We use the SMT solver again to solve the resulting formula with respect to the current solution *sol*. The solution *sol* defines a *partial instance* for this formula: it defines values for those variables of the formula that occurred in the path condition, i.e. $\texttt{SSA}(v_i, \pi_{i_0})$. Since the program point π_{i_0} is invalid, none of the executions of π_{i_0} gives a program state equal to the one defined by *sol*, and the above formula is unsatisfiable with respect to *sol*[4]. A solver capable of generating proofs (e.g. [8]) can be asked for a *proof of unsatisfiability* [28] – a formula weaker than the solved formula that is still unsatisfiable with respect to the given partial instance. Suppose that a formula f is unsatisfiable with respect to a partial instance i, and that, in the normal form, it consists of a conjunction of a set of clauses C, then the unsatisfiability proof will be a subset $C' \subseteq C$ which is still unsatisfiable with respect to i.

Consider again the above-mentioned example fragment along with the values `x=5` for the initial input, and `x'=5, y'=3, z'=18` for the final values. We can solve the formulas (1)-(3) with respect to these values. However, because the values do not represent a valid execution of the code, the solver cannot find a satisfying solution. Instead, it returns an unsatisfiability proof like $(x = X0) \wedge (X0 > 0 \implies y' = X0 + 5)$, which is equivalent to $(x > 0 \implies y' = x + 5)$. It highlights a small portion of the original formula that contradicts the given values, i.e. it is still unsatisfiable with respect to the values.

Intuitively, the unsatisfiability proof encodes the reason that the current solution *sol* is not consistent with the program. Because the proof is unsatisfiable with respect to *sol*, when conjoined with the current path condition, it prevents this invalid solution from ever being found again. The conjunction of the path condition and the proof constitutes the refined path condition and will be solved for a solution in the next iteration of our algorithm.

An ideal unsatisfiability proof is "minimal", meaning that any formula strictly weaker than the proof is satisfiable with respect to the partial instance. A min-

[4] Since the guard conditions evaluate to `true` given the current solution, leaving the `else` branches unconstrained does not affect the satisfiability of the formula.

1.	**int[] a = new int[3];**			
2.	**a[0] = System.in.read();**	[a[0] = 15]	[$a[0] \mapsto A0$]	
3.	**a[1] = System.in.read();**	[.., a[1] = 5]	[.., $a[1] \mapsto A1$]	
4.	**a[2] = System.in.read();**	[.., a[2] = 42]	[.., $a[2] \mapsto A2$]	
5.	**assert(a[0] > 0);**			$\{(A0 > 0)\}$
6.	**assert(a[1] > 0);**			$\{.. \wedge (A1 > 0)\}$
7.	**assert(a[2] > 0);**			$\{.. \wedge (A2 > 0)\}$
8.	**int sum = 0;**	[.., sum = 0]	[.., $sum \mapsto 0$]	
9.	**int i = 0;**	[.., i = 0]	[.., $i \mapsto 0$]	
10.	**if (i < 3) {**			$\{.. \wedge (0 < 3)\}$
11.	**if (sum = 0)**			$\{.. \wedge (0 = 0)\}$
12.	**sum = sum + a[i];**	[.., sum = 15]	[.., $sum \mapsto A0$]	
13.	**i = i + 1;**	[.., i = 1]	[.., $i \mapsto 1$]	
14.	**if (i < 3) {**			$\{.. \wedge (1 < 3)\}$
15.	**if (sum = 0)**			$\{.. \wedge \neg(A0 = 0)\}$
16.	sum = sum + a[i];			
17.	**i = i + 1;**	[.., i = 2]	[.., $i \mapsto 2$]	
18.	**if (i < 3) {**			$\{.. \wedge (2 < 3)\}$
19.	**if (sum = 0)**			$\{.. \wedge \neg(A0 = 0)\}$
20.	sum = sum + a[i];			
21.	**i = i + 1;**	[.., i = 3]	[.., $i \mapsto 3$]	
22.	**if (i < 3) {**			$\{.. \wedge \neg(3 < 3)\}$
23.	if (sum = 0)			
24.	sum = sum + a[i];			
25.	i = i + 1;			
	}}}}			
26.	**System.out.println(sum);**			

(a) (b)

Fig. 3. (a) Execution trace of the code for the given solution, (b) its symbolic execution.

imal proof guarantees that each iteration of our algorithm grows the path condition by only a minimal amount of information about the code. No proof-generating SMT solver, however, is guaranteed to produce minimal proofs. Nevertheless, experiments show that the generated proofs are small enough to be used in practice [24].

4 Example

Consider again the program of Figure 1, together with its SSA form and its PDG. The question was whether the secret value in `a[1]` could flow to the public output `sum` or not. As mentioned before, the path condition $PC(3, 14)$ given in Section 2 is satisfiable, indicating a potential illegal flow. But this is a false alarm, caused by the fact that the PDG cannot distinguish between different iterations of a loop. In fact, because the input values are expected to be positive, after the first iteration, we have `sum>0`, and thus the secret value `a[1]` is *not* added to `sum`, hence does not influence the public output.

Suppose that an SMT solver produces the following solution to the path condition: $\mathtt{a}[0] = 15, \mathtt{a}[1] = 5, \mathtt{a}[2] = 42, \mathtt{i}_1 = 0, \mathtt{i}_2 = \mathtt{i}_3 = 1, \mathtt{sum}_1 = \mathtt{sum}_2 = \mathtt{sum}_3 = \mathtt{sum}_4 = 0$, which is in fact a *false witness* for flow. This solution is checked for validity by executing the original code. The execution trace is given in Figure 3(a): executed statements are given in bold, and updates to the program state after each statement are given in square brackets.

During the execution, the program state is monitored and compared against the solution to determine invalid program points. Here the first invalid point is the loop entry. That is because, based on the semantics of the SSA form of Figure 1, the values $\texttt{i}_2 = 1$ and $\texttt{sum}_2 = 0$ must happen at the beginning of some loop iteration. But that is not the case, and thus this program point is invalid.

Based on the trace of Figure 3(a), a symbolic execution is performed as shown in Figure 3(b). The symbolic execution maintains (1) a mapping from variables to their symbolic expressions, and (2) a collection of all branch conditions taken. Figure 3(b) uses square brackets to represent the updates to the mapping after each statement is executed, and curly braces to represent branch conditions as added to the collection. The execution starts by introducing the symbolic constants $A0$, $A1$, and $A2$ for the initial values of the input variables, which are then used to compute subsequent expressions. Numerical expressions are simplified on the fly. The branch conditions are computed symbolically and denote which branch of a conditional has been taken. That is, any time an else branch is taken, the negation of the condition is stored.

At the end of the symbolic execution, a formula is generated to express the fact that $\texttt{sum}_2$ and $\texttt{i}_2$ must be the same as the symbolic values computed for `sum` and `i` at the beginning of some loop iteration. A conjunction of branch conditions is necessary for soundness. Conditions over constant values are evaluated on the fly. The resulting formula is given below:

$$
\begin{array}{lr}
(a[0] = A0)\ \wedge (a[1] = A1)\ \wedge (a[2] = A2)\ \wedge & (\textit{Initial symbolic values}) \\
\texttt{let}\ c = (A0 > 0) \wedge (A1 > 0) \wedge (A2 > 0)\ \texttt{and}\ c' = c \wedge \neg(A0 = 0)\ \texttt{in} & \\
\quad ((\texttt{sum}_2 = \texttt{if}\ c\ \texttt{then}\ 0\ \texttt{else ANY}) \wedge (\texttt{i}_2 = \texttt{if}\ c\ \texttt{then}\ 0\ \texttt{else ANY})) & (\textit{Line}\ 10) \\
\ \vee ((\texttt{sum}_2 = \texttt{if}\ c\ \texttt{then}\ A0\ \texttt{else ANY}) \wedge (\texttt{i}_2 = \texttt{if}\ c\ \texttt{then}\ 1\ \texttt{else ANY})) & (\textit{Line}\ 14) \\
\ \vee ((\texttt{sum}_2 = \texttt{if}\ c'\ \texttt{then}\ A0\ \texttt{else ANY}) \wedge (\texttt{i}_2 = \texttt{if}\ c'\ \texttt{then}\ 2\ \texttt{else ANY})) & (\textit{Line}\ 18) \\
\ \vee ((\texttt{sum}_2 = \texttt{if}\ c'\ \texttt{then}\ A0\ \texttt{else ANY}) \wedge (\texttt{i}_2 = \texttt{if}\ c'\ \texttt{then}\ 3\ \texttt{else ANY})) & (\textit{Line}\ 22)
\end{array}
$$

The intermediate variables c and c' are used only to improve readability of the formula. Note that, in this example, c and c' are equivalent and thus, all expressions use the same guard condition. The formula is then transformed to use logical implications rather than ternary `if` expressions. A proof-generating SMT solver is used to solve this formula with respect to the witness, which was $\texttt{a}[0] = 15, \texttt{a}[1] = 5, \texttt{a}[2] = 42, \texttt{i}_1 = 0, \texttt{i}_2 = \texttt{i}_3 = 1, \texttt{sum}_1 = \texttt{sum}_2 = \texttt{sum}_3 = \texttt{sum}_4 = 0$. Since the witness is spurious, the solver cannot find a solution. Instead, it produces the following proof of unsatisfiability:

$$\textit{Proof} \equiv ((a[0] > 0) \wedge (a[1] > 0) \wedge (a[2] > 0)) \implies ((\texttt{sum}_2 = 0 \wedge \texttt{i}_2 = 0) \vee \texttt{sum}_2 = a[0])$$

The proof is a consequence of the solved formula that is still unsatisfiable with respect to the given witness. Intuitively, it expresses the fact that if the program is executed (i.e. the assert conditions hold), the values of `sum` and `i` at the loop entry must both be 0 (for the very first loop iteration), or the value of `sum` must be the same as `a[0]` (for any subsequent loop iterations). Neither of these cases is true for the given witness, and that is why it is spurious. This proof is then conjoined with the original path condition to form the refined path condition.

$$\begin{aligned} PC'(3,14) &\equiv PC(3,14) \wedge Proof \\ &\equiv (\mathtt{a}[0] > 0) \wedge (\mathtt{a}[1] > 0) \wedge (\mathtt{a}[2] > 0) \wedge (\mathtt{i}_1 = 0) \wedge \\ &\quad\ (\mathtt{i}_2 = \mathtt{i}_3 = 1) \wedge (\mathtt{sum}_1 = \mathtt{sum}_2 = \mathtt{sum}_3 = \mathtt{sum}_4 = 0) \wedge \\ &\quad\ ((a[0] > 0) \wedge (a[1] > 0) \wedge (a[2] > 0) \\ &\qquad\qquad\qquad \implies (\mathtt{sum}_2 = 0 \wedge \mathtt{i}_2 = 0) \vee \mathtt{sum}_2 = a[0]) \end{aligned}$$

This refined path condition is again solved for a satisfying solution. This time the path condition is unsatisfiable, and thus, the analysis terminates with no solutions found, proving that there are no flows from `a[1]` to `sum`. Although in this example, the path condition became unsatisfiable after the first refinement, that is not always the case. In general, depending on the order in which the solver searches the space and the richness of the unsatisfiability proofs that it generates, several refinements might be needed to reach a conclusive outcome.

5 Related Work

Most IFC tools are based on security type systems [2,3,14,18,22], which opened the door into the whole field of language-based security. The Mobius project [2] developed a complete security infrastructure based on type systems and proof-carrying code. However, extending security type systems with a CEGAR approach seems difficult, as type systems do not generate logical formulae which can be refined as in our approach. On the PDG side, the TAJ project [25] implemented an IFC for full Java which scales to 500MLOC; but since TAJ is based on thin slicing, it does not discover implicit flow.

As classical noninterference is very restrictive, many information flow systems allow declassifications. Declassifications are controlled violations of legal flow. Our PDG-based IFC allows declassification at any PDG node, thus offering fine-grained control "where" declassification should take place. We have shown that PDG-based declassification respects monotonicity of release [12]. If path conditions are computed for declassification paths, they answer "when" a declassification happens. If a witness can be generated from the condition, this gives insight into the "what" of a declassification.

CEGAR-based program analysis has had tremendous success over the past few years. The software model checker SLAM [1], for example, analyzes C programs by iteratively refining a predicate abstraction of code, represented as a boolean program. BLAST [13] applies lazy abstraction and local refinements to achieve better performance. ARMC [19] uses a Prolog system with CLP extensions to perform predicate abstraction refinement. Although these tools have been successfully applied to substantial programs, they focus on checking code against temporal safety properties. We have previously used unsatisfiability proofs to refine abstractions of Java code in order to check precise data structure properties [24]. This paper builds on a similar approach of using unsatisfiability proofs, but in the context of information flow analysis. To our knowledge, this is the first time that CEGAR has been applied to IFC and software security.

6 Conclusion and Future Work

We presented the foundations of a new approach to IFC that improves the precision of our previous PDG-based analysis by incorporating a fully automatic refinement technique. The technique is a novel instantiation of CEGAR built on path conditions. It follows a solve-validate-refine loop to find witnesses for flow, check their validity, and eliminate the false ones. While our previous path condition-based IFC was already context-sensitive, flow-sensitive, and object-sensitive, it was purely static and could generate false alarms. The new approach uses constraint solving, concrete and symbolic executions, and unsatisfiability proofs to detect and eliminate false alarms, and thus to increase IFC's precision.

Currently, our PDG-based IFC is fully implemented, and a path condition generator for Java is available as a prototype. As mentioned above, the implementation supports declassification. The implementation of the new CEGAR-based approach, however, has just begun. Once a prototype implementation is completed, we can evaluate its precision and scalability on large programs. We expect the approach to be much more expensive than the PDG-based IFC alone, as it includes many solver queries and detailed dynamic and symbolic program runs. The run-time, however, is not really an issue for IFC where the analysis of a critical core may as well run overnight. We expect the approach to scale to a few kLOC – enough to check security-critical software cores. Note also that declassification is not yet integrated into our CEGAR approach.

We will explore two options for translating undecidable program expressions to the decideable solver logic. First, for many program constructs, such as 32-bit arithmetics, there are precise translations to bitvectors, which can be solved efficiently. Second, undecideable subexpressions can always be abstracted away using uninterpreted variables – but this may reduce precision again. Future work will investigate and evaluate these methods.

In case the scalability of the approach is shown to be insufficient, some techniques are possible to decrease the complexity of the generated constraints, and thus to reduce the time spent by the constraint solver. One such technique is to partition the symbolic execution into smaller pieces and introduce intermediate uninterpreted constants to simplify the expressions computed in each piece. Evaluating the effects of such optimizations is left for future work. The fundamental idea of a CEGAR-based IFC, however, opens the door to an IFC precision that cannot be achieved by a static analysis alone, neither type systems nor PDGs.

References

1. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.
2. G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E. Vétillard. Mobius: Mobility, ubiquity, security. objectives and progress report. In *TGC '06*, LNCS. Springer-Verlag, 2006.
3. G. Barthe and L. P. Nieto. Secure information flow for a concurrent language with scheduling. *Journal of Computer Security*, 15(6):647–689, 2007.

4. M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. The Barcelogic SMT solver. In *CAV '08*, pages 294–298, 2008.
5. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT solver. In *CAV '08*, pages 299–303, 2008.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00*, pages 154–169, 2000.
7. R. Cytron, J. Ferrante, B. Rosen, and et. al. Efficiently computing static single assignment and control dependence graph. *TOPLAS*, 13(4):451–490, 1991.
8. B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV '06*, volume 4144 of *LNCS*, pages 81–94, 2006.
9. D. Giffhorn and C. Hammer. Precise slicing of concurrent programs, an evaluation of precise slicing algorithms for concurrent programs. *JASE*, 16(2):197–234, 2009.
10. C. Hammer. *Information Flow Control for Java, A Comprehensive Approach on Path Conditions in Dependence Graphs.* PhD thesis, Universität Karlsruhe, 2009.
11. C. Hammer. Experiences with PDG-based IFC. In *International Symposium on Engineering Secure Software and Systems (ESSoS'10)*, pages 44–60, 2010.
12. C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *J. of Information Security*, 8(6):399–422, 2009.
13. T. A. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV '02*, pages 526–538, 2002.
14. S. Hunt and D. Sands. On flow-sensitive security types. In *POPL '06*, pages 79–90. ACM, 2006.
15. D. Jackson. Hazards of verification. In *Proc. Haifa Verification Conference*, volume 5394 of *LNCS*, pages 1–1, 2008.
16. J. Krinke. Context-sensitive slicing of concurrent programs. In *ESEC/FSE '03*, pages 178–187, New York, NY, USA, 2003. ACM.
17. J. Krinke. Program slicing. In *Handbook of Software Engineering and Knowledge Engineering*, volume 3: Recent Advances. World Scientific Publishing, 2005.
18. A. C. Myers. JFlow: practical mostly-static information flow control. In *POPL '99*, pages 228–241, New York, NY, USA, 1999. ACM Press.
19. A. Podelski and A. Rybalchenko. Armc: The logical choice for software model checking with abstraction refinement. In *PADL'07*, pages 245–259, 2007.
20. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.
21. M. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *SAS '09*, volume 5673 of *LNCS*, pages 3–18, 2009.
22. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL '98*, pages 355–364, San Diego, CA, January 1998.
23. G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *TOSEM*, 15(4):410–457, 2006.
24. M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. *Journal of Automated Software Engineering*, 14(1):87–121, 2007.
25. O. Tripp, M. Pistoia, S. Fink, M. Sridharan, and O. Weismani. TAJ: effective taint analysis of web applications. In *PLDI '09*, pages 87–97. ACM, 2009.
26. D. Wasserrab and D. Lohner. Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In *VERIFY-2010*, 2010.
27. D. Wasserrab, D. Lohner, and G. Snelting. On PDG-based noninterference and its modular proof. In *PLAS '09*. ACM, June 2009.
28. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker. In *DATE '03*, pages 10880–10886, 2003.