Experiences with PDG-based IFC

Christian Hammer

Purdue University*
cjhammer@purdue.edu

Abstract. Information flow control systems provide the guarantees that are required in today's security-relevant systems. While the literature has produced a wealth of techniques to ensure a given security policy, there is only a small number of implementations, and even these are mostly restricted to theoretical languages or a subset of an existing language.

Previously, we presented the theoretical foundations and algorithms for dependence-graph-based information flow control (IFC). As a complement, this paper presents the implementation and evaluation of our new approach, the first implementation of a dependence-graph based analysis that accepts full Java bytecode. It shows that the security policy can be annotated in a succinct manner; and the evaluation shows that the increased runtime of our analysis—a result of being flow-, context-, and object-sensitive—is mitigated by better analysis results and elevated practicability. Finally, we show that the scalability of our analysis is not limited by the sheer size of either the security lattice or the dependence graph that represents the program.

Key words: software security, noninterference, program dependence graph, information flow control, evaluation

1 Introduction

Information flow control is emerging as an integral component of end-to-end security inspections. There is very active research on how to secure the gap between access control and various kinds of I/O. Recent approaches to IFC are mainly based on special type sytems [20,21], and increasingly also on other forms of program analysis. This work reports on the experiences gained from our recent approach of the latter kind [14], and relates the insight gained with other tools based on type systems. The approach leveraged in this paper is based on *program dependence graphs* [8], more precisely *system dependence graphs* [15], which faithfully represent the semantics of a given program. The actual information flow validation is a graph traversal in the style of *program slicing*, which is a common program transformation on dependence graphs. The *(backward) slice* of a given statement contains all statements that may semantically influence

* This work was partially funded by Deutsche Forschungsgemeinschaft (DFG grant Sn11/9-2) at Karlsruhe University, Germany, where the experiments were conducted, and by the Office of Naval Research.

```
class A {
 int x;
 void set() { x = 0; }
 void set(int i) { x = i;}
 int get() { return x; }
}
class B extends A {
 void set() { x = 1; }
}
class InfFlow {
 PrintStream sysout = ...;
 void main(String[] a){
  //1. no information flow
  int sec = 0 /*P:High*/;
  int pub = 1 /*P:Low*/;
  A o = new A();
  o.set(sec);
  o = new A();
  o.set(pub);
  sysout.println(o.get());
  //2. dynamic dispatch
  if (sec==0 && a[0].equals(
    "007")) o = new B();
  o.set();
  sysout.println(o.get());
 }
}
```

Fig. 1. An example program for information flow control

that statement. Therefore, program slicing is closely connected to information flow control, which (in its simplest form, known as *noninterference*) checks that no secret input channel of a given program might be able to influence what is passed to a public output channel [11].

The major contributions of this work are:

– This paper presents the first dependence-graph-based IFC for full Java bytecode and its plugin integration into the Eclipse framework.
– Our IFC mechanism allows for succinct security policy specification as the number of required annotations is greatly reduced compared to traditional security type systems. This results a major improvement in practicability.
– The evaluation found that flow-, context-, and object-sensitive analysis pays off: While the analysis time increases compared to insensitive analyses, the results become significantly more precise, avoiding all sorts of false positives.
– Furthermore, the evaluation indicates that the analysis scales well to the security kernels we had in mind and that the scalability of our IFC analyses is not limited by either the size of the security lattice (which might be seen as a measure of the security policy's complexity), nor by the size of the program's dependence graph.

Previous work presented the theoretical foundations of dependence-graph-based IFC and described algorithms to efficiently implement these techniques. All these details are beyond the scope of this article, which focuses on presenting the framework for IFC in Eclipse and evaluating the effectiveness of these techniques. The interested reader is refered to our previous publications [12, 14] to find out all the details that cannot be covered in this work.

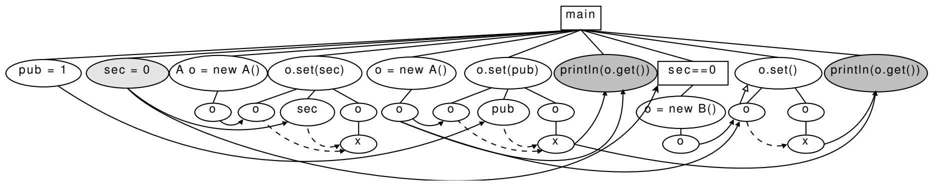


Fig. 2. SDG for the program in Figure 1

2 IFC Analysis

To illustrate dependence-graph-based IFC, Fig. 2 shows a (partial) SDG for Fig. 1. In the program, the variable `sec` is assumed to contain a secret value, which must not influence printed output. First a new `A` object is created where field `x` is initialized to `sec`. However, this object is no longer used afterward as the variable is overwritten ("killed") with a new object whose `x` field is set to `pub`. Note that there is no path in the dependence graph from the initialization of `sec` to the first `print` statement, which means that `sec` cannot influence that output, as SDGs contain all possible information flows. This example demonstrates that in our SDG-based analysis the `x` fields in the two `A` objects are distinguished (object-sensitivity), the side-effects of different calls to `set` are not merged (context-sensitivity), and flow-sensitivity kills any influence from the `sec` variable to the first `println`. An analysis missing any of these three dimension of sensitivity would reject this part of the program, thus producing a false positive.

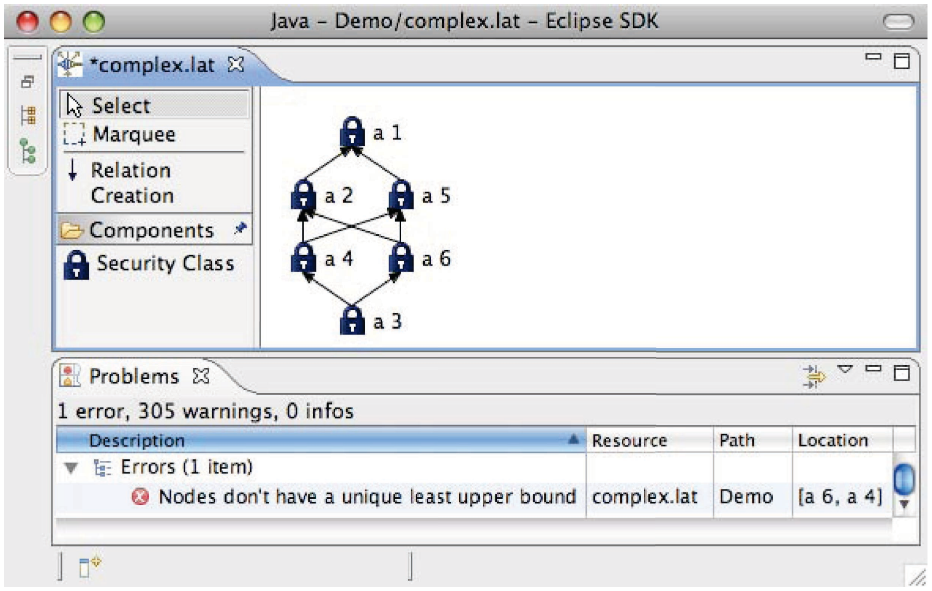
The next statements show an illegal flow of information: Line 22 checks whether `sec` is zero and creates an object of class `B` in this case. The invocation of `o.set` is dynamically dispatched: If the target object is an instance of `A` then `x` is set to zero; if it has type `B`, `x` receives the value one. Lines 22–24 are analogous to the following implicit flow:

```
if (sec==0 && ...) o.x = 0 else o.x = 1;
```

In the PDG we have a path from `sec` to the predicate testing `sec` to `o.set()` and its target object `o`. Following the summary edge one reaches the `x` field and finally the second output node. Thus the PDG discovers that the printed value in line 25 depends on the value of `sec`.

3 Eclipse Plugins for IFC

We have implemented SDG-based IFC analysis, including declassification, as described in our previous work [14]. The prototype is an Eclipse plugin, which allows interactive definition of security lattices, automatic generation of SDGs, annotation of security levels to SDG nodes via source annotation and automatic security checks. At the time of this writing, all these components are fully operational.



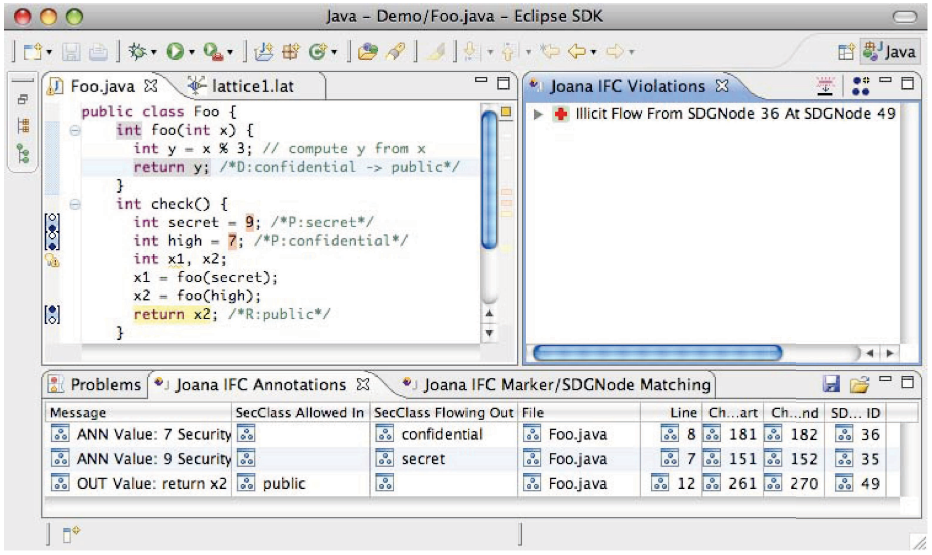
1 error, 305 warnings, 0 infos

Description	Resource	Path	Location
Errors (1 item)			
Nodes don't have a unique least upper bound	complex.lat	Demo	[a 6, a 4]

Fig. 3. The lattice editor in Eclipse with illegal graph

We implemented the lattice editor based on Eclipse's GEF graph editing framework. The lattice elements are represented as bit vectors [1, 9] to support fast infimum/supremum operators when checking for illegal information flow. It is worth noting that the algorithm of Ganguly et al. computes incorrect results without adding synthetic nodes into edges that span more than one level (where levels are defined in terms of the longest path between two nodes). The authors were not very specific concerning this restriction. Fortunately, these synthetic nodes can safely be removed from the lattice after conversion to a bit vector. Our editor attempts to convert the given graph to such a lattice. If this fails, the user is notified that the graph does not represent a valid lattice. Otherwise the lattice can be saved on disk for annotations. Figure 3 shows an example of a non-trivial graph with top element “a 1” and bottom element “a 3”. But the lattice conversion fails for this graph, as the elements “a 4” and “a 6” do not have a unique upper bound: Both “a 2” and “a 5” are upper bounds for these elements, which violates a crucial property of lattices. The problem view at the bottom displays a detailed message to the user that describes the violation. If one of the relation edges between those four elements were removed, a valid lattice would be generated, which can be leveraged for annotating the source code.

The IFC algorithm was implemented in a two-stage version: As the first step, the so-called *summary declassification nodes* are computed. If the dependence graph already contained standard *summary edges* [15], these need to be removed first, as they would disturb computation of summary declassification nodes. Still, generating summary edges during SDG generation was not in vain:



```
public class Foo {
    int foo(int x) {
      int y = x % 3; // compute y from x
      return y; /*D:confidential -> public*/
    }
    int check() {
      int secret = 9; /*P:secret*/
      int high = 7; /*P:confidential*/
      int x1, x2;
      x1 = foo(secret);
      x2 = foo(high);
      return x2; /*R:public*/
    }
```

```
Illicit Flow From SDGNode 36 At SDGNode 49
```

Message	SecClass Allowed In	SecClass Flowing Out	File	Line	Ch...art	Ch...nd	SD... ID
ANN Value: 7 Security		confidential	Foo.java	8	181	182	36
ANN Value: 9 Security		secret	Foo.java	7	151	152	35
OUT Value: return x2	public		Foo.java	12	261	270	49

Fig. 4. Example program (with missing declassification) in our Eclipse plugin

As summary declassification nodes can only arise between nodes that originally were connected by a summary edge, we can omit building the transitive closure of dependences for all nodes that are not the target of a summary edge. Algorithm 2 of [14] is therefore initialized with these actual-out nodes only. Note that this optimization does not improve the worst case complexity of the algorithm, but it reduces analysis time in practice (see section 5). As a second step, the IFC constraints are propagated through the backward slice of each output channel according to Algorithm 1 of [14] . Our implementation does not generate these constraints explicitly, but performs a fixed point analysis.

Figure 4 shows an example program but is missing a declassification in the `foo` method. The Eclipse plugin features a full-fledged view for annotations and security violations. User annotations are shown in the Joana IFC Annotations view at the bottom of the figure. The message shows the kind of annotation (ANN stands for provided security level, OUT for required security level, and RED for declassification with both). Next to the message, the R and P annotations are shown. The rest of the entries describe the annotated source position and the ID of the SDG node. Another View, called “Joana IFC Marker/SDGNode Matching” allows precise matching of the selected source to its respective SDG node according to the debug information provided in the class file. The last view, depicted on the right in Figure 4 lists all violations found by the last IFC checking. For the example program, a run of the IFC algorithm determines a security violation between nodes 36 ($P(36) = \mathit{secret}$) and 49 ($R(49) = \mathit{public}$)

because of the missing declassification in `foo`. When this declassification from *confidential* to *public* is introduced, no more illicit flow is detected.

4 Case Studies

As an initial micro-benchmark to compare our approach with type-based IFC, let us reconsider the program from Figure 1. Remember that PDG-based IFC guarantees that there is no flow from the `secure` variable (annotated $P(11) = \mathit{High}$) to the first output statement in line 20. Hence we analyzed the program from Figure 1 using Jif [18]. Jif uses a generalization of Denning's lattices, the so-called decentralized label model [19]. It allows to specify sets of security levels (called "labels" based on "principals") for every statement, and to attach a set of operations to any label. This is written e.g. $\{o_1 : r_1, r_2; o_2 : r_2; r_3\}$ and combines access control with IFC.

But note that the decentralized labels encode these access control features in an automatically generated standard security lattice and thus cannot overcome the imprecision of type-based analysis. As an example, we adapted the first part of Figure 1 to Jif syntax and annotated the declaration of `o` and both instantiations of `A` with the principal `{pp:}`. The output statement was replaced by an equivalent code that allowed public output. Jif reports that secure data could flow to that public channel and thus raised a false alarm. In fact, no annotation is possible that makes Jif accept the first part of Figure 1 without changing the source code.

4.1 A JavaCard Applet

As another case study for IFC we chose the JavaCard applet called `Wallet`.[1] It is only 252 lines long, but with the necessary API parts and stubs the PDG consists of 18858 nodes and 68259 edges. The time for PDG construction was 8 seconds plus 9 for summary edges on an Intel Core 2 Duo CPU with 2.33GHz and 2GB of RAM.

The Wallet stores a balance that is at the user's disposal. Access to this balance is only granted after supplying the correct PIN. We annotated all statements that update the balance with the provided security level *High* and inserted a declassification to *Low* into the `getBalance` method. The methods `credit` and `debit` may throw an exception if the maximum balance would be exceeded or if there is insufficient credit, resp. In such cases JavaCard applets throw an exception, and the exception is clearly dependent on the result of a condition involving balance. The exception is not meant to be caught but percolates to the JavaCard terminal, so we inserted declassifications for these exceptions as well. Besides this intended information flow, which is only possible upon user request and after verifying the PIN, our analysis proved that no further information flow is possible from the balance to the output of the JavaCard.

[1] http://www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard.html

Note that this JavaCard applet—while operating on a restricted variant of Java—leverages many features of this standard: In particular we analyzed about 85 static variables and included all control flow due to implicit exceptions into our analysis without the need to explicitly declare or catch these. Some of these features would at least require reengineering of source code, others are explicitly prohibited by Jif, so this benchmark cannot be certified with Jif. Again we find that the increased precision of dependence graph based analysis allows a more permissive language.

4.2 The Battleship Example

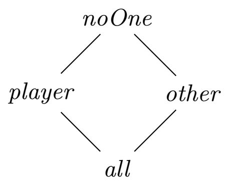


Fig. 5. The lattice for analyzing the battleship example

The previous experiments demonstrated that our new approach is more general than Jif, because we can analyze realistic programming languages (in principle all languages that compile to Java bytecode) and accept a larger number of secure programs due to increased precision. The next step in our evaluation will examine a native Jif program to get a direct comparison of practicability between these two systems. As a benchmark program, we chose the battleship example, which comes with every Jif installation and implements a non-GUI version of the popular battleship game. In this game, two players place ships of different lengths on a rectangular board and subsequently "bombard" random cells on the opponents board until one player has hit all the cells covered by adversary ships.

The source code of this program consists of about 500 lines plus the required libraries and stubs. These yield an SDG consisting of 10207 nodes and 77290 edges. For this example we use a standard diamond lattice, where $all \leq player \leq noOne$ and $all \leq other \leq noOne$ but neither $player \leq other$ nor $other \leq player$ (see Figure 5). This ensures that secret information of one player may not be seen by the other player and vice versa.

Before this example program could be analyzed by our IFC analysis, it had to be converted back to regular Java syntax. This included removal of all security types in the program, conversion of all syntactic anomalies like parentheses in throws clauses, and replacing all Jif peculiarities like its own runtime system. Most of this process required manual conversion. We annotated the ship placement strategy in the players initialization method with the security level

$P(n) = player$. The three declassification statements of the original Jif program are modeled as declassifications from *player* to *all* in our system as well. Then we annotated all parameters to `System.out.println` with $R(x) = all$, which corresponds to the original program's variable annotation.

```
 1 /**
 2  * Initialize the board by placing ships to cover numCovered coords.
 3  */
 4 void init/*{P:}**/(int/*{}**/numCovered) {
 5     // Here what we would do in a full system is make a call to
 6     // some non-Jif function, through the runtime interface, to
 7     // get the position of the ships to place. That function would
 8     // either return something random, or would implement some
 9     // strategy. Here, we fake it with some fixed positions for
10     // ships.
11     final Ship/*[{P:}]**/[] myCunningStrategy = {
12         new Ship/*[{P:}]**/(new Coordinate/*[{P:}]**/(1, 1), 1, true),
13         new Ship/*[{P:}]**/(new Coordinate/*[{P:}]**/(1, 3), 2, false),
14         new Ship/*[{P:}]**/(new Coordinate/*[{P:}]**/(2, 2), 3, true),
15         new Ship/*[{P:}]**/(new Coordinate/*[{P:}]**/(3, 4), 4, false),
16         new Ship/*[{P:}]**/(new Coordinate/*[{P:}]**/(5, 6), 5, true),
17         new Ship/*[{P:}]**/(new Coordinate/*[{P:}]**/(5, 7), 6, false),
18     };
19
20     Board/*[{P:}]**/board = this.board;
21     int i = 0;
22     for (int count = numCovered; count > 0 && board != null; ) {
23         try {
24             Ship/*[{P:}]**/newPiece = myCunningStrategy[i++];
25             if (newPiece != null && newPiece.length > count) {
26                 // this ship is too long!
27                 newPiece = new Ship/*[{P:}]**/(newPiece.pos,
28                                                count,
29                                                newPiece.isHorizontal);
30             }
31             board.addShip(newPiece);
32             count -= (newPiece==null?0:newPiece.length);
33         }
34         catch (ArrayIndexOutOfBoundsException ignored) {}
35         catch (IllegalArgumentException ignored) {
36             // two ships overlapped. Just try adding the next ship
37             // instead.
38         }
39     }
40 }
```

Fig. 6. Initialization method of a Player in Battleship

When we checked the program with this security policy, illicit information flow was discovered to all output nodes. Manual inspection found that all these violations were due to implicit information flow from the players initialization methods, more precisely, due to possible exceptions thrown in these methods. However, closer inspection found that all of these exceptional control flow paths are in fact impossible.

As an example consider the initialization method in Figure 6. If the program checks whether a SSA variable is null, and only executes an instruction involving this variable if it is not (cf. line 25), then no null-pointer exception may ever be thrown at that instruction. However, our intermediate representation currently does not detect that fact, even if two identical checks for null are present in the intermediate representation, one directly succeeding the other. Jif supports such local reasoning. For less trivial examples, where a final variable is defined in the constructor, and may thus never be null in any instance method, Jif requires additional annotation. We plan to integrate an analysis to detected such cases even in the interprocedural case [16]. Jif can only support non-local reasoning with additional user annotations.

Apart from null-pointer problems we found exceptional control flow due to array stores, where Java must ensure that the stored value is an instance of the array components, because of Java's covariant array anomaly. When a variable of an array type $a[\,]$ is assigned an array of a subtype $b[\,]$ where $b \leq a$, then storing an object of type a into that variable throws an `ArrayStoreException`. Here Jif seems to have some local reasoning to prune trivial cases (see lines 11-17 in Figure 6). Our intermediate representation does currently not prune such cases, however, with the pointer analysis results we use for data dependences, such impossible flow could easily be removed.

Lastly, for interprocedural analysis, we found that our intermediate representation models exceptional return values for all methods, even if a method is guaranteed to not throw any exception. Pruning such cases can render the control flow in calling methods more precise and remove spurious implicit flow. Jif requires user annotations for such cases, as all possibly thrown exceptions must either be caught or declared, even `RuntimeException`s, which do not have to be declared in usual Java.

Currently, our tool does not offer such analysis, so there are only external means to detect such spurious cases: Either by manual inspection, theorem proving (e.g. pre-/post-conditions), or path conditions [13]. After verifying that such flow is impossible, we can block the corresponding paths in the dependence graphs, and we do that with declassification. In contrast to normal declassifications, where information flow is possible but necessary, this declassification models the guarantee that there is no information flow. As future work, we plan to integrate analyses which prune impossible control flow to reduce the false positive rate and thus the burden of external verification.

After blocking information flow through exceptions in Player's initialization, our IFC algorithm proved the battleship example secure with respect to the described security policy. No further illicit information flow was discovered. During the security analysis, based on only four declassifications, 728 summary declassification nodes were created. This result shows that summary declassification nodes essentially affect analysis precision, as they allow context-sensitive slicing while blocking transitive information flow at method invocation sites. Instead they introduce a declassification that summarizes the declassification effects of the invoked methods. Note that the original Jif program contained about 150 annotations (in Figure 6 these are shown as gray comments), most of which are concerned with security typing. Some of these annotations model, however, principals and their authority. Still the number of annotations is at least an order

Table 1. Data for benchmark programs

Nr	Name	LOC	Nodes	Edges	Time	Summary
1	Dijkstra	618	2281	4999	4	1
2	Enigma	922	2132	4740	5	1
3	Lindenmayer	490	2601	195552	5	10
4	Network Flow	960	1759	3440	6	1
5	Plane-Sweep	1188	14129	386507	24	13
6	Semithue	909	19976	595362	24	33
7	TSP	1383	6102	15430	15	2
8	Union Find	1542	13169	990069	36	103
9	JC Wallet	252	18858	68259	8	9
10	JC Purse	9835	135271	1002589	145	742
11	mp	4750	271745	2405312	141	247

Table 2. Characteristics of the lattices in the evaluation.

	nodes	height	impure	bits
Lattice A	5	5	0	6
Lattice B	8	4	1	5
Lattice C	8	6	2	10
Lattice D	12	8	2	14
Lattice E	24	11	7	25
Lattice F	256	9	246	266

of magnitude higher than with our analysis. For a program that contains only 500 lines of code (including comments), this means that the annotation burden in Jif is considerable.

One of the reasons why slicing-based IFC needs less annotations than type systems is that side-effects of method calls are explicit in dependence graphs, so no end-label (which models the impact of side-effects on the program counter) is required, neither are return-value or exception labels. Those are computed as summary information representing the dependences of called methods.

Apart from these labels, Jif requires explicit annotations to verify any non-trivial property about exceptional control flow. In particular, many preconditions (e.g., non-nullness) need to be included into the program text instead of its annotations, e.g. explicit tests for null pointers or catch clauses, which are typically followed by empty handlers as in the example shown in Figure 6. Preconditions are therefore included as runtime tests to enable local reasoning. Such coding style is an ordeal from a software engineering perspective, as it impedes source comprehension and may conceal violated preconditions, which conflicts with Dijkstra's principle of the weakest precondition. What one really wants to have is verification that such cases cannot happen in any execution and thus do not need to be included into the source code.

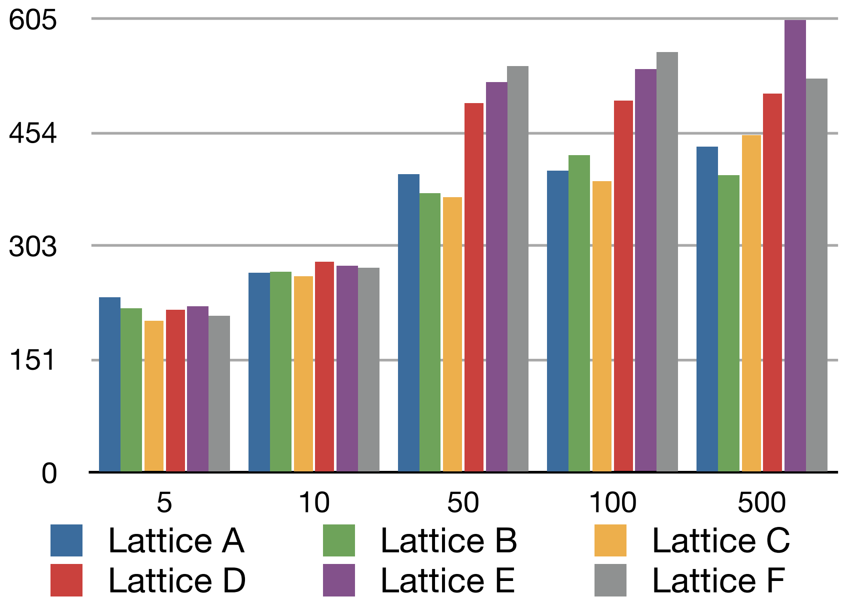


Fig. 7. Avg. execution time (y-axis, in s) of IFC analysis for the unionfind benchmark with different lattices and varying numbers of declassifications (x-axis)

5 Scalability

The previous sections demonstrated the precision and practicability of our approach. To validate the scalability of our new slicing-based information flow control, we measured execution times on a number of benchmarks with varying numbers of declassification and using lattices based on different characteristics. The benchmark programs are characterized in Table 1. We evaluated a benchmark of 8 student programs with an average size of 1kLoc, two medium-sized JavaCard applets and a Java application. The student programs use very few API calls, and for nearly all we designed stubs (for details see [12]) as to not miss essential dependences. The "Wallet" case study is the same as in section 4.1, the "Purse" applet is from the "Pacap" case study [5]. Both applet SDGs contain all the JavaCard API PDGs, native methods have been added as stubs. The program `mp` is the implementation of a mental poker protocol [3]. Again stubs have been used where necessary.

Table 2 shows the characteristics of the lattices we used in our evaluations: The first column displays the number of nodes in the lattice, the next column the maximal height of the lattice. The number of impure nodes in the lattice, which is shown in the next column, represents all nodes that have more than one parent in the lattice. The final column displays the number of bits needed in the efficient

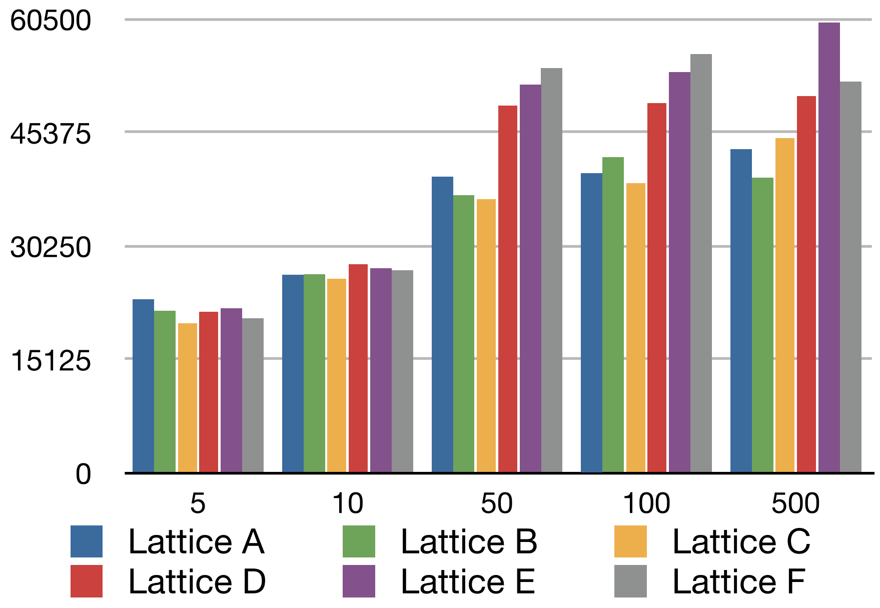


Fig. 8. Time for summary declassification nodes (in s) of unionfind with different lattices and varying numbers of declassifications (x-axis)

bitset encoding of Ganguly et al. [9]. This encoding allows near-constant[2] computation of infima (greatest lower bounds), which will turn out to be essential for our evaluation. The lattices for evaluation have been designed such that they cover different characteristics equally: Lattice A is a traditional chain lattice, lattice B is more flat and contains an impure node. Lattice F has been automatically generated by randomly removing edges from a complete subset lattice of 9 elements. Conversion to bitset representation is only possible for the Hasse diagram, i.e. the transitive reduction partial order, which is not guaranteed by random removal of order edges. So we included a reduction phase before bitset conversion. Interestingly, Table 2 illustrates that the bitset conversion usually results in a representation with size linear in the number of lattice nodes.

Figure 7 shows the average execution time of 100 IFC analyses (y-axis, in seconds) for the unionfind benchmark of Table 1 using the lattices of Table 2. We chose the unionfind benchmark here, as it had the longest execution time, and the other benchmarks essentially show the same characteristics. For all IFC analyses we annotated the SDGs with 100 random security levels as provided and required security level, respectively. Moreover, we created 5 to 500 random declassifications to measure the effect of declassification on IFC checking (shown on the x-axis). The numbers illustrate that our IFC algorithm is quite indepen-

[2] The infimum computation is in fact constant, but we need hashing to map lattice elements to bitsets.

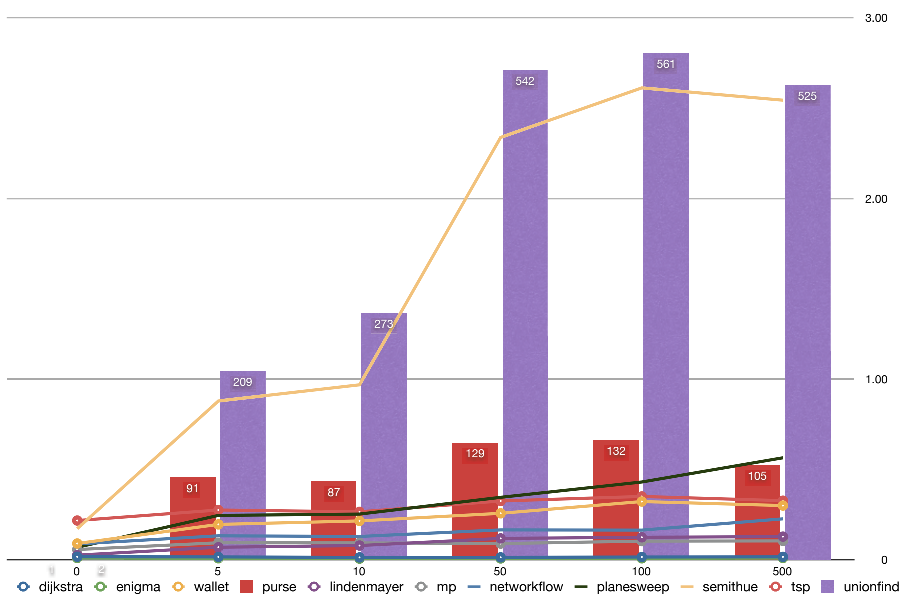


Fig. 9. Avg. execution time (y-axis, in s) of IFC analysis for all benchmark programs with the largest lattice and varying numbers of declassifications (x-axis). Bars use a different scale.

dent of the lattice structure and size. In particular, we got a sub-linear increase in execution time with respect to the lattice (and bitset) size. Apart from that, the increase with the number of declassifications is also clearly sub-linear, since the number of declassifications increases more than linear in our experiments (see y-axis). Figure 8 depicts the execution time for computing summary declassification nodes, which is a prerequisite for precise IFC checking (for details see [14, section 7]), therefore they have been acquired once for each combination of program, lattice, and declassifications. They were determined with the same random annotations as the numbers of Figure 7. Note that we did only compute summary information between nodes that were originally connected by summary edges. These numbers expose the same sub-linear correlations between time and lattice size or numbers of declassifications, respectively.

Figure 9 and 10 show the average execution time (y-axis, in seconds) of 100 IFC analyses and the time for summary declassification node computation, respectively, for all benchmark programs using the largest lattice and varying numbers of declassifications. Lines in this graph use the scale depicted on the right, while bars use a different scale, such that we included the numbers into each bar. For most programs, the analyses took less than a minute, with only semithue, purse, and unionfind requiring more time. Again, we found the correlation between execution time and number of declassifications sub-linear. In fact,

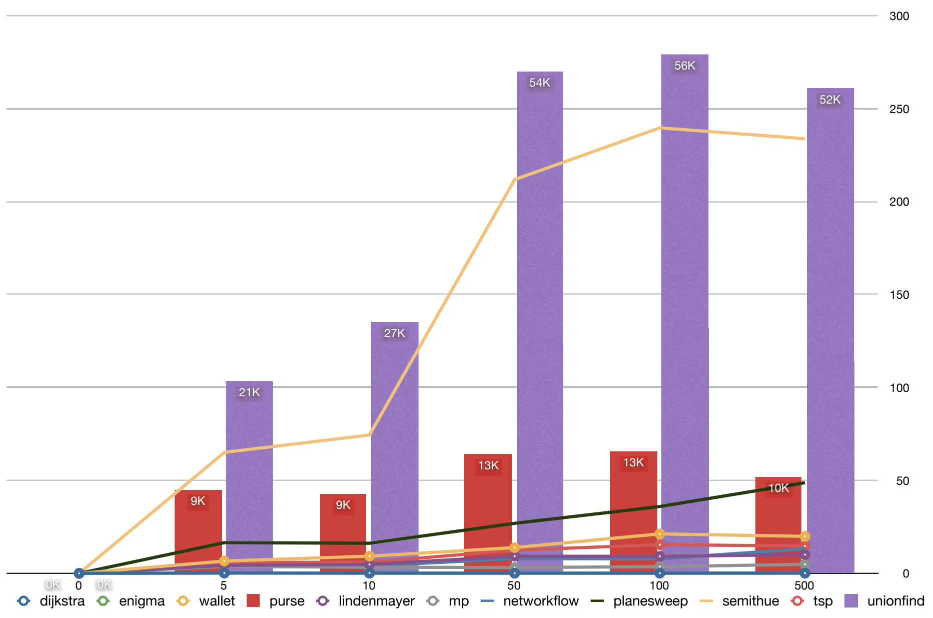


Fig. 10. Time for summary declassification nodes (y-axis, in s) for all benchmark programs with the largest lattice and varying numbers of declassifications (x-axis). Bars use a different scale.

the execution time for many benchmarks was lower with 500 declassifications than with 100. These numbers clearly illustrate the scalability of our information flow control analysis. There is no clear correlation between the number of nodes in the dependence graph and analysis time.

However, there seems to be a correlation between the number of edges in the SDG and the execution time. Unlike slicing, our IFC analysis is not linear in the number of SDG nodes and edges, but must find a fixed point in the constraint system with respect to the given lattice. Therefore, it may traverse a cycle in the SDG as often as the lattice is high, and when cycles are nested this effect may become even worse. Our current implementation does not circumvent these effects, so one observes that the programs with the most edges yield to a substantial increase in analysis time. But note that the largest program, mp, does not belong to the outliers but behaves good-naturedly. One reason might be the different program structure, which can also be seen from original summary edge computation (see Table 1), which is considerably lower than for other large programs. This program does—unlike JavaCard applets and our student programs—not have a big loop in the main method which may invoke nearly all functionality. Concluding, we assume that the program's structure plays a bigger role than the pure number of nodes or edges for analysis time.

While future work must evaluate the impact of standard slicing optimizations on this technique for faster fixed point computation, we think that 1 minute execution time, as observed by the majority of our test cases, is definitely reasonable for a security analysis. But even the three outliers require maximally 1.5 hours (including summary declassification analysis), which should be acceptable for a compile-time analysis that usually needs to be done only once.

5.1 Future Work

While we presented evidence for precision, scalability, and practicability, there is still room for further improvements: In particular, we expect that optimizations for slicing, e.g., as presented by Binkley et al. [6], apply to our information flow analyses as well. These techniques produce up to 71% reduction in runtime and thus significantly improve scalability. Further research must evaluate which of these techniques are applicable to information flow control. Apart from that, compositionality is given for type systems but other analyses, like pointer analysis which is a prerequisite for our dependence graphs, are usually not. For more scalability of our approach, we plan to investigate how to make dependence-graph-based IFC compositional.

6 Related Work

Research in information flow control has been predominantly theoretic during the last decade (cf. e.g. [17, 20, 21]) where proposed systems were not implemented at all. More recently, increasingly more approaches are at least implemented for a rudimentary language, often just a while-language. With Java being a mainstream language, several approaches have targeted a core bytecode language (e.g. [2, 4]), but essential features like exceptions, constructors, or unstructured control flow are often not taken into account. Smith and Thober [22] present a type inference algorithm for a subset of Java which ameliorates the immoderate annotation burden of type based IFC. In contrast, the work described in this paper supports full Java bytecode with unstructured control flow, procedures, exception handling, etc., and shows that it scales to realistic security kernels.

Genaim and Spoto [10] define an abstract interpretation of the CFG looking for information leaks. It can handle all bytecode instructions of single-threaded Java and conservatively handles implicit exceptions of bytecode instructions. The analysis is flow- and context-sensitive but does not differentiate between fields of different objects. Instead, they propose an object-insensitive solution that folds together all fields of a given class. In our experience [12], object-insensitivity yields too many spurious dependences. The same is true for the approximation of the call graph by class hierarchy analysis. In this setting, both will result in many false alarms.

Chandra and Franz [7] implemented a hybrid IFC framework, where Java bytecode is analyzed statically and the security policy is checked dynamically, which allows dynamic updates of the policy. However, dynamic enforcement imposes a slowdown factor of 2. To improve performance of dynamic label computation, they only allow fully ordered sets instead of the general security lattices used in this work. It is, however, not clear if fast infimum computation [9] is actually slower than their more restrictive scheme. In contrast to their system we currently only allows a fixed security policy with purely static checking, which induces higher compile time overhead in favor of zero runtime overhead.

For several years, the most complete and elaborate system for Java-like languages has been Jif [18]. As noted before, Jif is neither an extension nor a restriction of Java, but an incompatible variant. Therefore it requires considerable reengineering efforts to convert a standard Java program to Jif. The newest version offers confidentiality as well as integrity checking in the decentralized label model. Our system differs from Jif in that it directly analyzes Java bytecode without requiring any refactoring of the source code. It, too, allows both integrity as well as confidentiality checking, though that entails manual definition of the security lattice, while Jif's decentralized label model automatically generates an appropriate lattice from the annotations. Still, our experiments indicate that the increased analysis cost is in fact mitigated by a significantly lower annotation burden and elevated analysis precision.

7 Conclusion

This paper evaluates our novel approach for information flow control based on system dependence graphs as defined in our previous work [14]. The flow-sensitivity, context-sensitivity, and object-sensitivity of our slicer extends naturally to information flow control and thus excels over the predominant approach for information flow control, which is security type systems.

The evaluation section showed that our new algorithm for information flow control dramatically reduced the annotation burden compared to type systems, due to its elevated precision. Furthermore, empirical evaluation showed the scalability of this approach. While it is clearly more expensive than security type systems, the evaluation demonstrates that security kernels are certified in reasonable time. As this certification process is only needed once at compile time, even an analysis that takes hours is acceptable when it guarantees security for the whole lifetime of a software artifact. As a consequence, this paper makes recent developments in program analysis applicable to realistic programming languages. The presented system implements the first dependence-graph-based information flow control analysis for a realistic language, namely Java bytecode. While dependence graph based IFC is not a panacea in that area, it nevertheless shows that program analysis has more to offer than just sophisticated type systems.

References

[1] Aït-Kaci, H., Boyer, R., Lincoln, P., Nasr, R.: Efficient implementation of lattice operations. ACM TOPLAS **11**(1), 115–146 (1989). DOI 10.1145/59287.59293

[2] Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: POPL '06, pp. 91–102. ACM (2006). DOI 10.1145/1111037.1111046

[3] Askarov, A., Sabelfeld, A.: Security-typed languages for implementation of cryptographic protocols: A case study. In ESORICS'05, *LNCS*, vol. 3679, pp. 197–221. Springer (2005). DOI 10.1007/11555827_12

[4] Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference Java bytecode verifier. In: ESOP '07, *LNCS*, vol. 4421, pp. 125–140 (2007). DOI 10.1007/978-3-540-71316-6_10

[5] Bieber, P., Cazin, J., Marouani, A.E., Girard, P., Lanet, J.L., Wiels, V., Zanon, G.: The PACAP prototype: a tool for detecting Java Card illegal flow. In Java Card, *LNCS*, vol. 2041, pp. 25–37. Springer (2000). DOI 10.1007/3-540-45165-X_3

[6] Binkley, D., Harman, M., Krinke, J.: Empirical study of optimization techniques for massive slicing. ACM TOPLAS **30**(1), 3 (2007). DOI 10.1145/1290520.1290523

[7] Chandra, D., Franz, M.: Fine-grained information flow analysis and enforcement in a Java virtual machine. In: 23rd Annual Computer Security Applications Conference, pp. 463–475. IEEE (2007). DOI 10.1109/ACSAC.2007.37

[8] Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM TOPLAS **9**(3), 319–349 (1987). DOI 10.1145/24039.24041

[9] Ganguly, D.D., Mohan, C.K., Ranka, S.: A space-and-time-efficient coding algorithm for lattice computations. IEEE Trans. on Knowl. and Data Eng. **6**(5), 819–829 (1994). DOI 10.1109/69.317709

[10] Genaim, S., Spoto, F.: Information flow analysis for Java bytecode. In VMCAI 2005, *LNCS*, vol. 3385, pp. 346–362. Springer (2005). DOI 10.1007/b105073

[11] Goguen, J.A., Meseguer, J.: Unwinding and inference control. In Symposium on Security and Privacy, pp. 75–86. IEEE (1984). DOI 10.1109/SP.1984.10019

[12] Hammer, C.: Information flow control for Java - a comprehensive approach based on path conditions in dependence graphs. Ph.D. thesis, Universität Karlsruhe (TH), Fak. f. Informatik (2009). URN urn:nbn:de:0072-120494

[13] Hammer, C., Schaade, R., Snelting, G.: Static path conditions for Java. In: PLAS '08, pp. 57–66. ACM (2008). DOI 10.1145/1375696.1375704

[14] Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. Journal of Information Security **8**(6), 399–422 (2009). DOI 10.1007/s10207-009-0086-1

[15] Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM TOPLAS **12**(1), 26–60 (1990). DOI 10.1145/77606.77608

[16] Hubert, L.: A non-null annotation inferencer for Java bytecode. In: PASTE '08, pp. 36–42. ACM (2008). DOI 10.1145/1512475.1512484

[17] Hunt, S., Sands, D.: On flow-sensitive security types. In: POPL '06, pp. 79–90. ACM (2006). DOI 10.1145/1111037.1111045

[18] Myers, A.C., Chong, S., Nystrom, N., Zheng, L., Zdancewic, S.: Jif: Java information flow. URL http://www.cs.cornell.edu/jif/

[19] Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM TOSEM **9**(4), 410–442 (2000). DOI 10.1145/363516.363526

[20] Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003). DOI 10.1109/JSAC.2002.806121

[21] Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. Journal of Computer Security **17**(5), 517–548 (2009). DOI 10.3233/JCS-2009-0352

[22] Smith, S.F., Thober, M.: Improving usability of information flow security in Java. In: PLAS '07, pp. 11–20. ACM (2007). DOI 10.1145/1255329.1255332