On Automatic Placement of Declassifiers for Information-Flow Security

Dave King[†] Susmit Jha[‡] Trent Jaeger[†] Somesh Jha[*] Sanjit A. Seshia[‡]

[†] The Pennsylvania State University
University Park, PA, 16802
{dhking,tjaeger}@cse.psu.edu

[*] University of Wisconsin
Madison, WI, 53706
jha@cs.wisc.edu

[‡] University of California
Berkeley, CA, 94720
{jha,sseshia}@eecs.berkeley.edu

Abstract

Security-typed languages can be used to build programs that are information-flow secure, meaning that they do not allow secret data to leak. Declassification allows programs to leak secret information in carefully prescribed ways. Manually placing declassifiers to authorize certain flows of information can be dangerous because an incorrectly placed declassifier can leak far more secure data than intended. Additionally, the sheer number of runtime flows that can cause an error means that determining where to place declassifiers can be difficult. We present a new approach for constructing information-flow secure programs where declassifiers are placed such that no unintended leakage occurs. Leakage restrictions are specified using hard constraints and potential declassifier locations are ranked using soft constraints. Finally, the placement problem is submitted to a pseudo-Boolean optimizing SAT solver that selects a minimal set of declassifiers that prevent unauthorized data leakage. These declassifiers can be reviewed by the programmer to ensure that they correspond with acceptable declassification points: if not, new hard constraints can be added and the optimization framework can be reinvoked. Our experimental results indicate that our analysis suggests declassifiers that will cause no more leakage than those placed by programmers in a fraction of the time it would take to perform a manual analysis. This work provides a foundation for less expert programmers to build information-flow secure programs and to convert existing programs to be information-flow secure.

1 Introduction

Programs are increasingly being trusted to enforce complicated security policies on user data. However, it is not always clear whether or not a program maintains the security of its data. Incorrectly designed programs can leak their secret information to the outside world. Security-typed languages and their compilers can be used to build secure systems [5, 11], with successful compilation meaning that a program is information-flow secure and will not leak information to unauthorized users [7, 19].

However, information-flow security is an extremely restrictive property: even a simple password check routine leaks a single bit of information about whether the guess is the same as the stored password. Consequently, most systems release secret information through *declassifiers*, which are specially designated program points authorized to modify the security label on data [24].

The addition of declassifiers is currently a manual task that must be used with caution, as careless placement of a declassifier can release far more information than was intended. For example, a careless programmer may inadvertently release a password, rather than the hash of the password. Even the release of small amounts of information may accumulate, such that an attacker can deduce a secret. This danger is similar to the inference control problem in multilevel databases [13], where incorrectly classifying the security level of certain database queries allows malicious users to infer secret information from their supposedly-public requests.

Our goal is to develop an approach for placing declassifiers automatically and quickly at locations similar to those an expert programmer would select. We cast the problem of choosing a good set of declassifiers as a constraint optimization problem. First, a set of leakage restrictions eliminate illegal declassification choices. These *hard constraints* must be satisfied by any placement solution, so they remove solutions that are clearly unacceptable from a security viewpoint. Second, we use *soft constraints* to define an evaluation function that specifies a relative preference to each candidate location. Finally, we submit the placement problem to a pseudo-Boolean optimizing SAT solver that finds a minimal set of declassifiers that comply with the required leakage restrictions. The programmer can then either accept

or reject these declassifiers: if the placement is unsuitable, then the programmer can add more hard constraints to narrow the search. These proposed declassifers may also reveal a bug in the original source code, requiring the programmer to revise the program before re-running the analysis.

We make the following novel contributions in this paper:

- The first *constraint-based optimization approach* for automatically placing declassifiers to resolve illegal information-flows in an application.
- The definition and combined use of both *hard and soft constraints* to precisely limit and rank the locations where declassifiers can be placed.
- A *provably optimal encoding* of information-flow constraints with potential declassifications into SAT constraints that are mostly Horn-SAT, the latter being a linear-time solvable subclass of SAT.

We present experimental results on Java programs demonstrating that our approach synthesizes declassifiers automatically, quickly, and in locations that will cause no more leakage than those placed by programmers in a fraction of the time it would take to perform a manual analysis.

The remainder of this paper is organized as follows: in Section 2, we outline the main challenges addressed by our problem. In Section 3, we present theoretical background on security-typed languages and information-flow constraints. In Section 4, we outline the placement constraints that we used for our analysis. We give details for augmenting the information-flow constraints generated by a program with potential fixes and converting them into psuedo-Boolean SAT constraints in Section 5. We discuss the results of our experiments in Section 6, review related work in Section 7, and conclude in Section 8.

2 Problem

In this section we examine in detail some issues with placing declassifiers in code. We present code for MASTERMIND, a game played between two players, `Alice` (the code-breaker) and `Bob` (the code-maker). Before the game begins, `Bob` sets the code, a sequence of colors. During the game, `Alice` repeatedly guesses a sequence of colors; to each guess, `Bob` responds with how close the guess is to the code. For example, if the code is *Red, Blue, Green, Red* and the codebreaker guesses *Blue, Blue, Yellow, Green*, then codebreaker gives the feedback: "one color is in the right position, and one color is correct but in the wrong position". Figure 1 shows Java code that implements MASTERMIND, with certain data tagged by security labels, as belonging to the codebreaker or the codemaker.

From the description of the game, it is clear that when run, MASTERMIND will leak information from Bob to Alice. In order to allow these information leaks, we must add

```
1  int[]{Bob} code;   // Bob's secret code
2  final int CORRECT_COLOR, CORRECT_COLOR_AND_POS;
3
4  public int[]{Alice} guess(int[]{Alice} guess) {
5      int[] returnArray = null;
6      if (guess.length == code.length) {
7          returnArray = new int[guess.length];
8          int[] used = new int[guess.length];
9          int k = 0;
10         for(int j = 0; j < code.length; ++j) {
11             int guessResult = −1;
12             if (guess[j] == code[j]) {
13                 used[j] = 1;
14                 guessResult = CORRECT_COLOR_AND_POS;
15             }
16             else for(int i = 0; i < code.length; ++i)
17                 if (guess[j] == code[i] &&
18                         used[i] != 1) {
19                     used[i] = 1;
20                     guessResult = CORRECT_COLOR;
21                     break;
22                 }
23             if (guessResult != −1)
24                 returnArray[k++] = guessResult;
25         }
26     }
27     return returnArray;
28 }
```

Figure 1. Mastermind Code Example

declassifiers to the code. Careful placement of declassifiers is essential: incorrectly placed declassifiers may leak more information than intended. Through careful analysis of the code and use of existing tools to explain information-flow errors, the programmer can see that information flows from `Bob` to `Alice` in several ways:

- **Flow 1**: Whether the returned array is `null` or not is dependent on the guess being the same length as the code. (`code.length` flows to `returnArray`)
- **Flow 2**: `returnArray` is modified a number of times equal to the length of the code. (`code.length` flows to `j` and then flows to `returnArray`)
- **Flows 3 and 4**: The contents of the returned array correspond with information based on the `code`. (`code[j]` flows to `guessResult` at both lines 14 and 20 and then flows to `returnArray`)
- **Flow 5**: The number of non-zero entries in the returned array depends on information from the `code`. (`code.length` flows to `j` and then flows to `returnArray`)

A programmer can manually add declassification statements around points that cause the above information flows: the comparisons in lines 6, 10, and the `guessResult` variable on lines 23 and 24. However, this requires a large amount of effort in understanding the code, understanding

the information flows involved, and understanding the best locations for declassifiers.

Our main observation is that inserting declassifiers into code can be framed as an optimization problem. with two parts. First, determining which locations in the code should be considered for declassification, and second, determining a minimal set of those locations that correct the information-flow errors in the program.

Not every location in the code is suitable for declassification. For example, we should not declassify `code[j]`, as it is a member of an array that has been explicitly tagged as secret. We can view these requirements as *leakage restrictions*: secure data should not be leaked in certain ways. In our system, these become *hard constraints* that restrict potential declassifier locations. Once the candidate declassifiers have been chosen, we rank the remaining locations using *soft constraints*: these involve factors where we would prefer to declassify at one location instead of another. For example, we may prefer to declassify `guessResult` at line 24 rather than the conditional at line 17, because leaking the variable `guessResult` will reveal less information about the protected `code` array. Declassifying at line 17 would leak information about which element of the code is equal to the current element of the guess, while declassifying at line 24 only leaks information that there *is* a match somewhere.

In order to determine a minimal set of declassifier locations that fix the illegal information flows in the code, we encode the information flow problem as a SAT optimization problem. After we have applied both the hard and soft constraints, we modify the set of information-flow constraints generated by the application to include these potential declassifications. These are then converted into pseudo-Boolean SAT constraints and solved by MiniSat+ [9].

Figure 2 summarizes the key elements in our approach:

- Our analysis engine, a modification of the Jif compiler [20], generates information-flow constraints for a security-annotated program.
- We define hard and soft constraints to limit and rank the candidate declassification locations (Section 4).
- We convert the information-flow constraints and candidate declassifications to pseudo-Boolean SAT constraints using a specialized constraint encoder (Section 5). These are solved by the MiniSat+ pseudo-Boolean solver.
- The results from the SAT solver are translated back into information-flow constraints and the selected declassifications are reported to the user (summarized in Section 6).
- If these declassifiers are not suitable, the programmer can add new hard constraints to the system in order to refine the generated candidate declassifiers. In the event that a declassifier reveals a programming error in the program, the programmer can revise the code as necessary, using information-flow error reporting tools to assist as necessary [14].

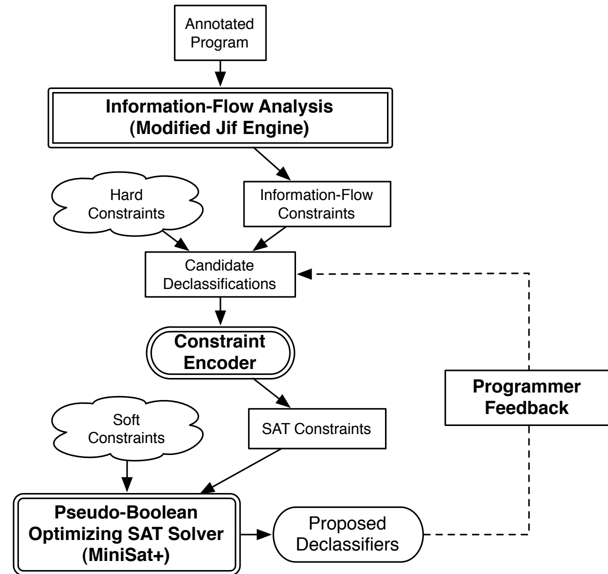

Figure 2. Overview of the Declassification Placement System

3 Background

Most security-typed language compilers check information-flow security by constraint solving. We present some background on this topic below. Though there are a number of security-typed languages, the implementation of existing compilers rely on the basic principles presented here.

3.1 Information-Flow Security

Information-flow security ensures that a program will not leak secret information when executed. However, full information-flow security is very restrictive, as real systems can leak data through timing channels, power attacks, and runtime exceptions [23]. In this paper we only focus on *explicit* and *implicit* flows of information between program variables.

An explicit flow of information from one variables to another occurs when information dependent on one variable is written into another. For example, `i` := `j`, `i` := `j` / 2, `i` := (`j` + `k`) % 5 are each explicit flows of information from `j` to `i`. An implicit flow of information is a flow where an observer can gain information about one variable by observing an assignment to another. For example, in the as-

signment **if** (`i` == 0) **then** `m` := 3, an observer watching `m` can deduce that if `m` is changed to 3, then `i` must be 0.

3.2 Information-Flow Constraints

Consider a lattice $\mathcal{L}$ over the set of security labels. Let L denote a security label and α be a variable from a countably infinite set of variables $\mathcal{V}$. Information-flow constraints c involve comparisons between label expressions τ, and have the following grammar:

$$\tau ::= \alpha \mid L \mid \tau_1 \sqcup \tau_2 \qquad c ::= \tau_1 \leq \tau_2$$

If τ is either a variable α or a label L, we say that τ is an *atom*. Let X be a set of information-flow constraints. Define $\mathrm{Vars}(c)$ to be the variables α that occur in c, and $\mathrm{Vars}(X)$ be the variables that occur in X. We say a constraint without variables $c \equiv \tau_1 \leq \tau_2$ is *satisfied* if $\mathcal{L} \models \tau_1 \leq \tau_2$; i.e., the $\leq$ relation holds between τ_1 and τ_2 when they are interpreted over lattice $\mathcal{L}$. We say that X is *satisfiable* if there exists a mapping $\varphi : \mathcal{V} \mapsto \mathcal{L}$ such that for each $c \in X$, $\mathcal{L} \models \varphi(c)$.

Almost all of the label constraints generated by security-typed language compilers such as Jif are *definite*, meaning that they have either the form $\tau \leq \alpha$ or $\tau \leq L$. Constraints of this form can be solved in polynomial time by a standard worklist algorithm [22]. Jif also generates a small number of *generalized* label constraints, meaning that the right-hand side contains a join of two or more atoms.

3.3 Security-Typed Languages

Security-typed languages provide compilers that can verify whether or not code is information-flow secure. At present, there are two mature security-typed language compilers; Jif [20], a variant of Java, and Flow Caml [21], a variant of Caml Light. To check the information-flow security of a piece of code, security-typed languages generate information-flow constraints, and then verify that these are satisfiable. For example, the assignment `j` := `k` generates the label constraint {`k`} <= {`j`} (read as: `k` is less secure than `j`), where {`k`} and {`j`} are label variables representing the security level of the program variables `k` and `j`. If `k` has been assigned `Public` security and `j` is given `Secret` security, then this label constraint is satisfied.

Implicit flows are forbidden by keeping track of data that affects the program counter. Returning to the above conditional, **if** (`i` == 0) **then** `m` := 0, when the program evaluates the conditional, the program counter is raised to {`i`}, as executing the statement `m` := 0 is dependent on the value of the variable `i`. This assignment then generates the label constraint {`i`} <= {`m`} (`i` is less secure than `m`).

The analysis we use in this paper expands the label analysis provided by the Jif compiler to be *context-sensitive*. This addition allows us to run our analysis on existing Java code without individually labeling the security behavior of each method.

4 Generating Candidate Locations for Declassifiers

In this section we describe the methods we used to place declassifiers in the code. We first narrow down potential declassifier locations using *hard constraints*: these are absolute restrictions on what places in the code cannot contain declassifiers. Next, we rank the remaining locations using *soft constraints*. We begin with some details about the declassification expressions that we place in code.

4.1 Declassifiers

We adopt the declassification system used in the security-typed language Jif. A declassifier consists of an expression and a target label; if `e` is an expression, `declassify(e,L)` declassifies the security on the value of `e` to `L` after evaluating `e`. However, as `e` must be evaluated, the security of its value after declassification is still affected by the program counter; this will be important later when we are introducing declassifications to the information-flow constraint set. Though Jif provides a mechanism to declassify the program counter, our experience has shown that these are rarely used and declassifying an expression is almost always preferred. Though our framework is general enough to handle these types of fixes, we restrict our attention to placement of expression declassifiers.

4.2 Sources and Sinks

The hard and soft constraints we present to determine declassification are motivated by the concepts of security-relevant sources and sinks and how they interact. A *source* is an expression where security-relevant data enters the program from the outside of the system; this could be from an API call, data stored in a library, or a variable that has been explicitly tagged by the programmer. A *sink* is an expression where data is passed to an explicitly-labeled program location.

From a broad perspective, secure data enters the system through specific security sources, is operated on by the instructions in the program, and is then passed back into a sink. An information-flow violation occurs when data with security label L is written into a sink with security label L' and $L \not\leq L'$. To resolve this illegal information flow, we must either rewrite the code to remove this information flow (which might cause the program to no longer function as necessary) or place a declassifier somewhere along the path from source to sink. We focus on the second option.

We use the information-flow constraints generated by our system to determine if the value of one expression flows to another. Let X be the set of definite information-flow constraints for a program. For each expression e, there is a variable α_e and an information-flow constraint $c_e \equiv \tau \leq \alpha_e$ in X. To determine whether or not the program permits an illegal flow of information through e, we build a *value-flow graph* G_X from the information-flow constraint set X.

Definition 4.1. Let G_X be the directed graph with vertices $\text{Vars}(X) \cup \mathcal{L}$ and edges E, where:

- $(\alpha, \beta) \in E$ if there exists $\tau \leq \beta \in X$ such that $\alpha \in \text{Vars}(\tau)$.
- $(L_i, \alpha) \in E$ if there exists $\tau_0 \sqcup L_i \leq \alpha \in X$.
- $(\alpha, L_i) \in E$ if there exists $\tau_0 \leq L_i$ such that $\alpha \in \text{Vars}(\tau)$.

Intuitively there is an edge between two vertices in the value flow-graph if the security value of one might affect the other. Let $v_0 \rightsquigarrow v_1$ in G_X if G_X contains a path from vertex v_0 to v_1. We write $e \rightsquigarrow f$ (e flows to f) if there is a path from α_e to α_f in G_X. If this is the case, then modifying the value of the variable α_e may modify the value of the variable α_f, and thus the security level of the value of f. We write $e \rightsquigarrow L$ if the graph contains a path from α_e to the security label L.

Sources and sinks are represented by the vertices labels L_i. We say that e is a source for L if $(L, \alpha_e) \in E$, and e is a sink for L if $(\alpha_e, L) \in E$). A label is a source if it has edges leaving it, while it is a sink if it has edges arriving at it. A program contains an illegal flow of information if there exist i, j such that $L_i \rightsquigarrow L_j$ and $L_i \not\leq L_j$.

4.3 Example: Value Flow in Mastermind

Returning to the MASTERMIND example from Figure 1, `guess` is a source for `Alice`, `code` is a source for `Bob`, and both `returnArray` and the method return statement are sinks for `Alice`. Initially, every expression in the code can be declassified to every possible label. Our security lattice contains labels for `Alice`, `Bob`, and `Public`, and the file contains 87 separate expressions in only 23 lines of code. We therefore begin with 261 potential declassifications. We first remove declassifiers for expressions that are always constant, leaving us with 232 candidate declassifications. To further narrow the declassification space, we consider other types of expressions and argue that they should never be declassified.

Consider each occurrence of the array `code`, belonging to `Bob`. As it has been explicitly labeled, we should not declassify it or its contents: we want to leak as little information as possible. The arrays `guess` and `returnArray` are also labeled, and so we should not declassify them either.

Observe that there are no sinks of label `Bob` or `Public` in the application; all of the sinks are of type `Alice`. As no expression can flow to `Bob` and `Public` in the value-flow graph, we can remove candidate declassifications to these target labels.

Finally, observe that there are many expressions contained inside larger expressions; for example, on line 10, `code.length` occurs inside of the larger conditional `j < code.length`. In this case, `code.length` only flows to `j < code.length`, and so declassifying this larger expression would have the same effect as declassifying `code.length` and `j` separately as both flow to a sink. This corresponds to the intuition that information leaked by an expression e is at most equal to the information leaked by the subexpressions of e.

After applying each of these hard constraints, the number of candidate declassifiers is reduced from 232 to 23. We remove 42 declassifiers by not declassifying sources, 52 by preventing declassification to labels that are not sinks for those expressions, and 117 by not declassifying expressions that only affect a larger expression.

4.4 Hard Constraints

We now present the hard constraints that we use for declassifier placement. These eliminate candidate expressions that should not be declassified from a system security perspective as well as assist the SAT solver by removing declassifiers that will never be part of an acceptable solution. These are only some of the possible hard constraints to restrict declassifier placement: we could design other ones adapted from other suggested patterns of declassification from the literature [3, 5, 11].

No declassification of sources: Do not declassify an expression e if e is a source for label L.

This requirement ensures that we will not modify the security of fields and variables that have already been explicitly labeled with a different security policy. This may cause certain applications to be unfixable; e.g., if a variable is explicitly labeled as `Secret` and sent to a `Public` sink. However, this type of situation can be detected with existing information-flow error detection tools [14]; if a error trace contains no constraints that are fixable, the system can warn the programmer or assign a high penalty to declassifying that expression (see the next section).

Only declassify an expression to labels that it flows to: Only declassify an expression e to L if there exists a sink of label L such that $e \rightsquigarrow L$.

This hard constraint prevents considering of declassifications to L when e does not flow to a sink of label L.

Do not declassify dominated expressions: Suppose $e \rightsquigarrow f$ and for all other x such that $e \rightsquigarrow x$, $f \rightsquigarrow x$ (e is dominated in the flowgraph by f). If f can be declassified

(i.e. none of the other hard constraints apply), then e should not be declassified.

This hard constraint prevents declassifying subexpressions: for example, we would not consider declassifying `j` if `j` was contained inside the expression `j` < `code.length`.

These hard constraints are a few simple rules for restricting declassifier locations. In practice we have found that these eliminate most spurious declassification sites. Programmers can also add hard constraints to our system based on feedback from previous outputs of the program; for example, a hard constraint can eliminate considering declassification inside a method that should not contain declassifiers.

4.5 Soft Constraints

Once we have applied the hard constraints to eliminate potential declassifiers, we may still have a large number of candidate declassification locations. For example, the situation occuring in Figure 3 is a common one: e affects both x and y, and their values eventually flow to a sink of label L. To resolve this program, we need to either declassify e, or both x and y, but our hard constraints do not eliminate either possibility. We use *soft constraints* to rank the remaining declassifications.

We would ideally rank expressions based on the number of bits that declassifying an expression would leak. However, accurately determining this amount is a topic of ongoing research; the main difficulty is determining the influence of loops [16, 17]. While promising, these analyses have not yet been applied to fullscale Java code. The optimization framework presented in this work is independent of our particular choice of soft constraint, and so as quantitative information-flow analyses improve, they can be incorporated as soft constraints. We could also add soft constraints to quantify other desireable properties, such as code maintainability.

For this experiment, we use a simple heuristic assigning the cost of declassifying an expression e based on the percentage of other expressions that e would flow to. Our results suggest that this simple heuristic generally performs well, but it could certainly be improved; for example, assigning a large penalty to expressions inside a loop.

Prefer declassifications that affect less of the code: To give a weight to an expression e, we assign e a weight correlated with the percentage of other expressions that e might affect. Let $RS(\alpha_e)$ be the subgraph of G_X reachable from α_e. We assign the weight of expression e as follows:

$$w_e = |V(RS(\alpha_e))|/|V(G_X)|$$

A declassification with a lower weight is preferred. We then multiply w_e by an integer constant C and round to receive an integer weight for the declassification (currently C = 100). The constant C must be large enough to distinguish between expressions, but small enough so that the pseudo-Boolean optimization is still efficient.

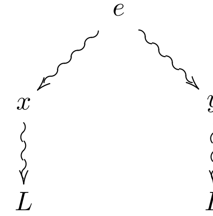

Figure 3. A situation that requires soft constraints to choose between candidate declassification locations. To resolve the information-flow errors, we need to assign a preference between declassifying e, x, and y.

As an example from the MASTERMIND code, the conditional `guess.length` == `code.length` on line 6 affects 94.8% of the code, while the comparison `used[i]` != -1 on line 18 only affects 20.5% of the code. Therefore, declassifying the length conditional has a weight of 95, while declassifying the array comparison only has a weight of 21.

Returning to Figure 3, if declassifying e has a weight of 50, while declassifying each of x and y has a weight of 30, we will prefer to place one declassifier at e (total cost: 50) rather than two at x and y (total cost: 60), even though the value of e may not explicitly flow to a sink. Soft constraints therefore permit declassification of more information than the application actually releases. While this is unfortunate, our experience is that many source-to-sink errors can be resolved by placing one declassifier. For example, in the Purse application in Section 6, six declassifiers resolved over sixty illegal flows. If certain expressions should not be declassified (for example, instances of a `PrivateKey` class), we can prevent this by adding new hard constraints to our system.

5 Constraint Optimization for Declassifier Placement

We now detail the translation from the generated information-flow constraints into Boolean constraints. The key steps in our technique are (1) *encoding declassification choices with Boolean variables* (2) *constraint simplification*, and (3) *Boolean encoding*. We first introduce the effect of potential declassifications into the constraint set (Section 5.1). We show that our encoding is asymptotically optimal (Corollary 5.5) and that the constraints generated by this procedure are mostly Horn-SAT and 2-SAT (Section 5.4). After describing the conversion to SAT constraints, we

show how to use weights assigned by soft constraints to formulate minimal placement of declassifications as a pseudo-Boolean optimization problem.

For clarity of presentation, we first describe our technique for definite constraints which are of the form $\tau \leq \alpha_e$, where τ is join of security labels and α_e is some security label. These are the most common form of constraints as mentioned in Section 3. In Section 5.6, we show how our technique can be used to encode generalized constraints which are of the form $\tau_1 \leq \tau_2$, where τ_1, τ_2 are a join of security labels.

5.1 Introducing Declassifications

Once we have narrowed down and ranked candidate declassification locations using hard and soft constraints, we must modify the constraint set to model the effect of these potential fixes. We model the choice of whether or not to introduce the ith declassifier by a Boolean variable d_i. If c is an information-flow constraint, *declassification constraints* dc have the form:

$$dc ::= d \implies c \mid \neg d_1 \wedge \cdots \wedge \neg d_n \implies c$$

The first form of declassification constraint can be read as "if d is true, than c holds," while the second can be read as "if none of $d_1, \ldots, d_n$ holds, then c holds."

An expression e is associated with a variable α_e and a definite constraint c_e of the form $\tau \leq \alpha_e$. Here, τ is a join of the security labels that affect the value of e; one of these labels will be pc_e, the program counter at e. As expression declassifiers do not remove explicit flows, declassifying the value of e to label L will replace the constraint $\tau \leq \alpha_e$ with the constraint $L \sqcup pc_e \leq \alpha_e$.

If we can potentially declassify e to labels $[L_1, \ldots, L_n]$, then we generate Boolean declassification variables $d_{e,1}, \ldots, d_{e,n}$ and replace the label constraint c_e in the constraint set with the following declassification constraints:

$$\begin{aligned} d_{e,1} &\implies pc_e \sqcup L_1 \leq \alpha_e \\ &\cdots \\ d_{e,n} &\implies pc_e \sqcup L_n \leq \alpha_e \\ \neg d_{e,1} \wedge \neg d_{e,2} \wedge \cdots \wedge \neg d_{e,n} &\implies \tau \leq \alpha_e \end{aligned}$$

For example, if $d_{e,1}$ is 1, meaning that e is declassified to L_1, then $pc_e \sqcup L_1 \leq \alpha_e$ must hold. If each of the $d_{e,i}$ are 0, then $\tau \leq \alpha_e$ must hold. Note that as long as each declassification variable has a non-negative weight, no minimal solution will choose $d_{e,i} = 1$ and $d_{e,j} = 1$ for $i \neq j$, as this would unnecessarily raise the level of α_e to $L_i \sqcup L_j \sqcup pc_e$.

5.2 Encoding Information-Flow Constraints as SAT Constraints

After the addition of Boolean declassification variables, we have a constraint problem that involves both Boolean and security label-valued variables. We show in this section how we can efficiently encode the problem entirely into the Boolean domain.

There are two steps in our solution - *simplification* and *Boolean translation*. The simplification reduces the constraints to those involving only two labels. For example, if a constraint $\alpha \sqcup L \leq \gamma \in X$, then there will be an edge from both α to γ and from L to γ in the value-flow graph. After simplification, each constraint corresponds to a unique edge in G_X.

After the constraints have been simplified, we adapt the Rehof-Mogensen (RM) technique [22] to construct a Boolean encoding of the constraints. The RM technique considers definite inequality constraints (the right-hand-side is either a variable or a constant) involving monotone functions over a finite lattice and shows that they can be reduced to Horn-SAT, which is solvable in polynomial time. However, the constraints generated by introducing potential declassifications are not definite due to the presence of Boolean declassification variables and the corresponding constraints. Thus, the RM technique can not be directly applied here. Instead, we give an efficient Boolean translation which reduces information flow constraints to *mostly Horn-SAT* constraints.

5.2.1 Constraint Simplification

The following is the key theorem allowing the simplification of the constraints to comparisons between two labels. The proof is straightforward and follows from the definitions and properties of precedence ($\leq$) and the LUB ($\sqcup$) operator.

Theorem 5.1. *$A_1 \sqcup \cdots \sqcup A_r \leq A_0$ if and only if $A_i \leq A_0$ for all i such that $1 \leq i \leq r$.*

From the above theorem, it follows that any constraint $d \implies A_1 \sqcup \cdots \sqcup A_r \leq A_0$ is equivalent to the set of r constraints

$$\{d \implies A_i \leq A_0 \mid \forall i\; 1 \leq i \leq r\}$$

If the above simplification is used for all constraints, we have reduced the problem of encoding the information flow constraints with potential declassifications to encoding constraints of the form $d \implies A \leq B$, where A and B are atoms. In case there was no declassification in the original constraint, d can be assumed to be $\mathtt{true}$. In the following section, we show how to encode constraints of form $d \implies A \leq B$ into a Boolean form.

5.2.2 Translation to SAT

The key idea in the translation to SAT is to build a Galois connection between the security lattice $\mathcal{L}$ and the lattice 2^n similar to the RM technique [22]. This will translate these inequalities over $\mathcal{L} = \{L_1, \dots, L_n\}$ to a set of inequalities over 2^n. The set of inequalities over 2^n is then a Boolean satisfiability (SAT) problem.

We define the Galois connection [6] using the same two adjoint functions as in RM technique [22]. The basic idea is to encode an element x of the lattice $\mathcal{L}$ as a Boolean vector containing the relationship betwen x and each element of $\mathcal{L}$.

Definition 5.2 (Translation Function)**.** The translation function $\phi : \mathcal{L} \to 2^n$ is defined as:
$\phi(x) = (1, b_1, \dots, b_n)$ where $b_i = 1$ if and only if $L_i \le x$
$\phi(\bot) = (0, 0, \dots, 0)$

Definition 5.3 (Inverse of Translation Function)**.** The inverse translation function $\psi : 2^n \to \mathcal{L}$ is defined as:

$$\psi((b_0, b_1, \dots, b_n)) = \bigsqcup \{L_i | b_i = 1\}$$

Both adjoint functions defined above are monotone and inverses: $x = \psi(\phi(x))$ and $v \le \phi(\psi(v))$. Let $\phi_i(x) = 1$ iff $L_i \le x$ (i-th component of $\phi(x)$).

This encoding of lattice into the Boolean domain requires $O(n^2)$ bits, as each element of an n-sized lattice is translated into a Boolean vector of length $n + 1$. We show that this encoding is *asymptotically optimal*. Theorem 5.4 shows that the number of lattices with n security labels is at least $2^{(n/2)^2}$.

Theorem 5.4. *The number of distinct lattices with n elements is $\Omega(2^{(n/2)^2})$.*

Proof. (by induction on number of elements)
Base Case: The number of distinct lattices constructed with 2 elements is $2 = 2^{(2/2)^2}$.
Induction Hypothesis: Let the number of lattices with k elements be at least $2^{(k/2)^2}$.

Let us take an arbitrary lattice with k elements. We count the number of ways it can be extended to a distinct lattice with $k+1$ elements by adding a new $(k+1)th$ element. We perform the extension by selecting a subset S from the set of k elements and making the $(k+1)$th element less than the elements in S. Clearly, each such extension will create a distinct lattice as all the $k+1$ elements are distinct.

Further let the order relation in the lattice with k elements be the set of tuples $O = \{(l_i, l_j) | l_i \le l_j\}$. The new additions to order relation O would be the set $\{(l_{k+1}, l_i) | l_{k+1} \le l_i$ and l_i is in $S\}$. Clearly, since there is no member of order relation O of the form (l, l_{k+1}), transitivity and anti-symmetric nature is still preserved.

Hence, a lattice with $k+1$ elements can be constructed from a lattice of k elements by adding a new element in at least as many ways as the number of subsets of k elements, that is, 2^k. Thus, the number of lattices with $k+1$ elements for $k \ge 1$ will be at least

$$2^{(k/2)^2} \cdot 2^k = 2^{(\frac{k}{2})^2 + k} = 2^{(\frac{k^2+4k}{4})} > 2^{(\frac{k+1}{2})^2}$$

The result thus follows by induction. □

Corollary 5.5. *Any Boolean encoding of a lattice with n elements requires $\Omega((n/2)^2)$ bits.*

Our encoding technique uses n^2 bits, which is just four times more than the proven lower bound.

5.2.3 Translating Simplified Constraints

We now show how to use the definitions of Galois connection defined above to translate the simplified constraints described in Section 5.2.1. Given a simplified constraint system X on lattice $\mathcal{L}$ of size n over the set of variables $V = \{\alpha_1, \alpha_2, \dots, \alpha_m\}$, we take the following steps:

1. **Extension of ϕ**: We extend ϕ to be defined over variables: let $\phi(\alpha_i) = (\alpha_{i0}, \alpha_{i1}, \alpha_{i2}, \dots, \alpha_{in})$. Intuitively, $\alpha_{ij} = 1$ means that $L_j \le \alpha_i$.
2. **Consistency Constraints on Variables:** For each variable α_i, we create a set of consistency constraints on the value α_i can take on. For each pair of elements of $\mathcal{L}$ such that $L_j \le L_k$, we add the SAT constraint $\alpha_{ij} \le \alpha_{ik}$.
3. **Information-Flow Constraints**: For each constraint of the form $A \le B$ and each i from 0 to n, we add the constraint $\phi_i(A) \le \phi_i(B)$.
4. **Declassification Constraints**: To translate a declassification constraint of the form $d \implies A \le B$, for each i from 0 to n, we generate the SAT constraint $d \implies \phi_i(A) \le \phi_i(B)$.

In the above, $\alpha_{ij} \le \alpha_{ik}$ is equivalent to $\neg\alpha_{ij} \vee \alpha_{ik}$. After having mapped to the 2^n space, the above definite inequalities obtained from constraints without declassification are Horn clauses. These are also 2-SAT clauses involving only two literals. In case of declassification, $d \implies \phi_i(A) \le \phi_i(B)$ is equivalent to $\neg d \vee \neg\phi_i(A) \vee \phi_i(B)$. Here, the generated constraint is a Horn clause if and only if d is a positive literal. Therefore, for each possible declassification, this procedure generates one Horn clause and one non-Horn clause.

5.3 Example: Mastermind Proposal and Encoding

After applying the hard and soft constraints, our system proposes declassifying the expression `j < code.length` to `Alice` on line 10 in Figure 1. The constraint associated

with this expression is $NV_{21} \sqcup NV_{18} \leq NV_{22}$, and the program counter at this point is pc_{18}. We introduce a new declassification variable d_7 and propose the declassification constraints:

$$\begin{array}{rcl} d_7 & \Longrightarrow & \text{Alice} \sqcup pc_{18} \leq NV_{22} \\ \neg d_7 & \Longrightarrow & NV_{21} \sqcup NV_{18} \leq NV_{22} \end{array}$$

We next apply simplification, giving us the constraint set:

$$\begin{array}{ll} d_7 \implies \text{Alice} \leq NV_{22} & d_7 \implies pc_{18} \leq NV_{22} \\ \neg d_7 \implies NV_{21} \leq NV_{22} & \neg d_7 \implies NV_{18} \leq NV_{22} \end{array}$$

Our lattice contains 4 elements: `Public`, `Alice`, `Bob`, and `Secret`. Therefore, applying the encoding function ϕ, results in the constraint set that contains for each $i = 0, \ldots, 3$:

$$\begin{array}{c} \neg d_7 \vee \neg\phi_i(\text{Alice}) \vee \phi_i(NV_{22}) \\ \neg d_7 \vee \neg\phi_i(pc_{18}) \vee \phi_i(NV_{22}) \\ d_7 \vee \neg\phi_i(NV_{21}) \vee \phi_i(NV_{22}) \\ d_7 \vee \neg\phi_i(NV_{18}) \vee \phi_i(NV_{22}) \end{array}$$

We therefore we have 8 Horn clauses and 8 non-Horn clauses corresponding to the declassification variable d_7.

5.4 Properties of Generated SAT Constraints

In this section, we analyze the nature of satisfiability problem obtained after the above reduction and show that it is very amenable to analysis; the constraints that we generate through our encoding will be very efficient to solve.

Consider first the case for information-flow constraints X without any declassification constraints. We will generate at most $r|X|$ simplified constraints where r is the maximum number of atoms in a single constraint in X. For each simplified constraint, we then generate $|\mathcal{L}|$ SAT constraints, one for each lattice element. Thus, we will create $r|\mathcal{L}||X|$ SAT constraints in translating X into a SAT problem. As, the lattice can have at most $|\mathcal{L}|^2/2$ order relation between the lattice elements, the number of consistency constraints during step 2 is at most $|\mathcal{L}|^2|V|$. Therefore, the total number of generated constraints is at most $r|\mathcal{L}||X| + |\mathcal{L}|^2|V|$. In the absence of any declassification, all these constraints will be Horn-SAT and 2-SAT.

Now, we extend the analysis to constraints with declassifications. If we have d declassification constraints, then:

- We will have $r|\mathcal{L}||X| + rd|\mathcal{L}| + |\mathcal{L}|^2|V|$ SAT clauses. This is two SAT constraints for each declassification constraint.
- Of these constraints, $r|\mathcal{L}||X| - rd|\mathcal{L}| + |\mathcal{L}|^2|V|$ will be 2-SAT.
- Finally, $r|\mathcal{L}||X| + |\mathcal{L}|^2|V|$ will be Horn-SAT, while only $rd|\mathcal{L}|$ will not be Horn-SAT.

It has been observed in literature on test generation for circuits [15] that SAT solvers can be efficiently used to generate test cases, as two-thirds of the SAT clauses generated from circuits are 2-SAT. Our experimental results indicate that, in general, our hard constraints reduce the set of candidate declassifiers enough so that our constraints are almost always more than two-thirds 2-SAT. We revisit this in Section 6.

5.5 Minimizing Declassifications as a Pseudo-Boolean Optimization

Now, we extend the solution of checking the satisfiability of the constraints and generating one such set of declassifications which will make the constraints satisfiable to the solution of finding the set of declassifications with minimal total cost. Since the solution of the decision problem involves using a satisfiability solver, the minimization problem can be solved using pseudo-Boolean constraint optimization [2].

The goal of minimizing the total cost of declassifications can be expressed as a pseudo-Boolean optimization problem in the following way. If w_i is the cost of a declassification d_i in X, we wish to find a satisfiable assignment $\mathcal{A}$ to the declassification variables such that $\mathsf{Cost} = \sum_{i=0}^{d} w_i \mathcal{A}(d_i)$ is minimized, subject to the generated SAT constraints.

Thus, the information flow constraint solver simplifies the constraints and then reduces them to an input to a pseudo-Boolean optimization solver which is given Cost as the objective to be minimized. The resulting assignments to declassification variables d_i ensure that the allowed declassifications will have minimum total cost.

The problem of finding a set of declassifications with minimum total cost is NP-complete. It follows from the NP-completeness of the problem of finding a minimum set of constraints that lead to an information leak [14]. The latter problem can be formulated as a special case of the former in which there is one candidate declassifier per constraint and a unit weight associated with each candidate declassifier. Thus, the above formulation as a pseudo-Boolean optimization problem is a natural one, and it also leverages the advances in SAT solving made in recent years. We have used MiniSat+ [9] as the pseudo-Boolean optimization solver in our experiments.

5.6 Encoding Generalized Constraints

We now describe how our approach can be extended to encode generalized constraints. As mentioned before, the Jif compiler generates a very limited number of constraints of the more general form $\tau_1 \leq \tau_2$, where τ_2 is the join of more than one label; these are generally created by termina-

tion restrictions on a procedure call. Such a constraint can be decomposed into the conjunction of a definite constraint and an equality, as shown below:

$$\tau_1 \leq \tau_2 \iff \tau_1 \leq \alpha \wedge \alpha = \tau_2$$

The definite constraint $\tau_1 \leq \alpha$ can be encoded using the technique described in Section 5.2.

We use the definition of join to encode $\alpha = \tau_2$ as pseudo-Boolean constraint. By the definition of joins,

$$\tau_2 \leq \alpha \wedge \forall \beta \, (\tau_2 \leq \beta \implies \alpha \leq \beta)$$

Recall from the definition of the encoding function ϕ that if $\phi(L) = (1, b_1, \ldots, b_n)$ and $\phi(L') = (1, b'_1, \ldots, b'_n)$, then

$$L \leq L' \iff \forall i \, b_i = 1 \implies b'_i = 1$$

The following theorem summarizes our encoding technique.

Theorem 5.6. *Let $\phi(\alpha) = (1, \alpha_1, \ldots, \alpha_n)$. Then, $\alpha = \tau_2$ if and only if $\tau_2 \leq \alpha$ for α such that $\sum_i \alpha_i$ is minimized.*

The minimization is over Boolean variables α_i (introduced during item 1 of Section 5.2.3) and hence is a pseudo-Boolean linear cost function.

Therefore, each generalized constraint can be encoded as two satisfiability constraints and one linear cost function to be minimized. The cost function is over the new variable α introduced to denote the join τ_2; therefore, the sets of variables mentioned in the newly introduced cost functions and the original pseudo-Boolean cost function $\sum_{i=0}^{d} w_i d_i$ are disjoint. Summing the original cost function with the newly introduced cost functions, we arrive at a single linear cost function that can be minimized to solve the declassifier placement problem with generalized constraints.

6 Evaluation

In this section we describe the results of running our analysis on several existing programs. We find that the SAT constraints generated by our encoding can be solved efficiently, and investigate the declassifiers chosen as optimal by the pseudo-Boolean solver. In general, the chosen declassifiers are placed in reasonable locations. We conclude by revisiting the analysis regarding the nature of generated SAT constraints from Section 5.4 and find that our experimental results agree with our statement that most of the generated SAT constraints are either Horn-SAT or 2-SAT. Our results are shown in Table 1. We ran our analysis with an Intel Pentium D with a 3 GHz CPU and 2 gigabytes of memory to generate these results.

Our main experimental results are:

- For Jif programs, our analysis selected declassifiers in the same locations as those manually placed by expert programmers, except in one case where it placed declassifiers in an alternative location that leaked an equivalent amount of information.
- For Java programs, our analysis selected a small number of declassifiers relative to the overall size of the code. In instances where programs already contained an authentication or authorization mechanism, our system selected these as optimal declassification locations.
- The generated SAT problems contained over 95% Horn-SAT clauses and generally a percentage of 2-SAT clauses over 75%. These percentages were inversely correlated with the ratio of candidate declassifiers to information-flow constraints.

6.1 Mastermind

Using a version of Jif expanded with error tracing [14], we found three initial errors in MASTERMIND, corresponding to the three sinks introduced in Figure 1. A limitation of this information-flow reporting mechanism is its inability to display more than one error message per sink. In order to resolve this problem, at least three declassifiers must be placed; one to resolve each error. However, MiniSat+ selected four declassifiers required for the application to be information-flow secure, corresponding to each of the flows in the program as introduced in Section 2.

- (Resolve Flows 1/2) declassify `guess.length` == **this**.`code.length` (line 6)
- (Resolve Flows 3/4) declassify `guessResult` != -1 (line 23)
- (Resolve Flows 3/4) declassify `guessResult` (line 24)
- (Resolve Flow 5) declassify `j` < **this**.`code.length` (line 10)

This is a minimal set of declassifiers that we can place in Figure 1 for information-flow security without refactoring the code by introducing new variables; e.g., creating a variable `codeLength` to store the declassified value of `code.length`.

6.2 Jif Programs

We ran our analysis on parts of two Jif programs. In general, we found that our system placed declassifications in the same or equivalent places to those placed by expert programmers.

Mental Poker: Mental poker is a cryptographic protocol allowing two or more parties to play a fair game of poker without a trusted third party and without revealing any information about their cards [3]. Each party uses a cryptographic matrix to encrypt the deck and verify that the other

Application	SLOC	Info-Flow Constraints	Candidate Declassifiers	Selected Declassifiers	SAT Constraints	SAT Variables	SAT Time (s)
Mastermind	38	172	23	4	1716	170	0.05
Wallet	127	796	79	3	4895	686	0.17
JPMail (MIME)	< 1189	1957	399	2	16434	1432	6.86
Mental Poker 1	< 1500	2357	571	2	20520	3059	39.18
Mental Poker 2	< 1500	3498	932	4	31173	4693	407.54
Purse	5781	34576	5536	6	298008	35829	10.31

Table 1. The results from our experiments. The first and second column contains the application name and the source-lines-of-code (SLOC) in the application; i.e. code not including whitespace and braces. The third column contains the number of information-flow constraints. The fourth and fifth column contains the number of candidate declassification sites and the number of declassifiers that were selected by our tool. The sixth and seven columns contain statistics about the generated SAT problem. The eighth column contains the time it took MiniSat+ to find an optimal solution.

parties are not cheating. In the implementation, the player owns secret data that is eventually encrypted, signed, and sent to a public distributed notarization chain.

We ran our analysis on two of the API functions in the Mental Poker program. Our system selected two types of declassifiers: (1) declassifiers involved in a secure computation, (2) declassifiers involved with the communication system. Declassifiers of type (1) were placed at program points where privately-labeled secret data was moved into a publicly-labeled encrypted data structure, while declassifiers of type (2) occurred where data that was labeled as secret was sent to the communication system. However, declassification in the Jif version of Mental Poker was handled by a helper data structure that would declassify an entire data structure, along with its contents, making our results not directly comparable. While our declassifiers leak the same information, the Jif implementation of Mental Poker places its declassifiers closer to the API layer. For the first API function, our system selected 2 declassifiers out of 571 candidate declassifiers. For the second API function, our system selected 4 out of 932 candidate declassifiers.

JPMail: JPMail is a prototype mail client written in Jif [11]. It ensures that the body of emails will only be revealed to the sender and receiver as it passes through the implemented mail subsystem. JPMail stores secure messages in a class containing secret data; when these messages are written out to MIME format, they are converted to having the security level of the channel that the message is being sent out at. If this security level not above the level of the message in the security lattice, the message is encrypted. Initially, our system suggested declassifying the body of the email before any encryption was done. This occurred because our labeling did not distinguish whether or not the destination of the message was above or below the current security of the message. We could expand our analysis to handle this case by adding runtime labels to our encoding of the lattice. When we changed the JPMail code to always encrypt a message, our system suggested placing declassifiers around the 2 calls to encryption functions (DES and AES) out of 399 candidate declassification locations. These correspond exactly to the declassifiers that were originally placed in JPMail. We could also use soft constraints that preferred declassifiers around calls to cryptographic primitive [27].

6.3 Java Programs

We ran our analysis on two Java Card programs and generally found that our analysis placed declassifiers near existing code mechanisms designed to enforce some variant of information-flow safety.

Wallet: `Wallet` is a small application used to teach beginners how to use the Java Card API [1]. It maintains a PIN number that protects the card from unauthorized tampering. We labeled the PIN number as `Secret` and the output as `Public`. Our system selected 3 declassifiers out of a total of 79 candidate declassifiers: one declassification of checking the PIN against a guess (during validation), and two later checks as to whether or not the PIN had been validated yet.

Purse: `Purse` is an application designed for use by the PACAP information-flow secure Java Card system [4]. It implements an electronic purse with a balance of money that can be credited, debited, and exchanged into different currencies. We labeled Purse corresponding to an integrity lattice[2], protecting data structures stored by the Purse as `Trusted` and the labeling the input as `Untrusted`. The application suggested 6 integrity endorsers out of 5536 candidate endorsers: 5 of these corresponded to a call to an access control table, and 1 of them corresponded to checking an entered PIN.

[1] Available at `http://developers.sun.com/mobility/javacard/articles/intro/`

[2] Integrity is dual to confidentiality. In an integrity lattice, `Trusted` is the lowest element and `Untrusted` is the highest. Declassifying in an integrity lattice is referred to as *endorsing*, as it increases the trust on a piece of data. Similarly an integrity declassifier is an *endorser*.

6.4 Discussion

In general, we found that our analysis performed well on a number of varied codebases. In instances where MiniSat+ took a large amount of time, it quickly found a solution that was within 10% of the optimal solution and then spent the remainder of the time trying to marginally improve that solution. For better performance, many pseudo-Boolean solvers, including MiniSat+, can be run in an anytime mode, thus permitting the user to trade optimality of the solution for a faster SAT running time.

Each of our examples was checked with a small security lattice (3 elements for most, 4 elements for MASTERMIND). Increasing the size of the lattice will increase the number of constraints and variables in the generated SAT problem. However, increasing the lattice should not affect the number of candidate declassifications, as our first hard constraint from Section 4.4 ensures that these will scale only with the number of sinks of different labels.

These analyses were run with more permissive label checking than Jif: specifically, we disabled implicit flows occuring from Java-specific runtime exceptions such as null pointer exceptions or array out of bounds exceptions. While our analysis supports these implicit flows, these flows either cannot be solved using declassifiers or would require declassifying far too much data. Future work will expand our system to support fixes other than declassification as part of a generalized automatic refactoring process.

Our experimental results agree with the calculations in Section 5.4. Table 2 provides some statistics about the SAT constraints that we generate with our encoding. We find, as expected, that applications with a lower percentage of declassifiers generate a more efficient SAT problem. In particular, while the Purse application is the largest application that we ran our tool on, the SAT encoding of information-flow constraints contained a very high percentage of Horn-SAT and 2-SAT constraints, and so it performed faster than smaller examples. The hard constraints are the major factor affecting the total percentage of the information-flow constraints that can be declassified. In particular, not declassifying completely dominated expressions reduces the set of candidate declassifications by a factor of at least three.

Application	Horn-SAT %	2-SAT %
Mastermind	98.70	79.12
Wallet	98.61	87.69
JPMail	98.70	62.35
Mental Poker 1	98.55	78.13
Mental Poker 2	98.44	76.67
Purse	99.05	85.11

Table 2. The percentage of generated constraints that are Horn-SAT or 2-SAT constraints. The higher the percentage of a SAT problem that is Horn-SAT or 2-SAT, the more efficiently it can be solved.

7 Related Work

Modern SAT solvers have recently been applied to a wide range of program analysis and synthesis problems, including bug detection [8, 28], lock allocation [10], and program synthesis based on user provided sketches [26]. Of this work, our approach is most closely related to the work on lock allocation by Emmi et al. [10], who also use a pseudo-Boolean SAT-based optimizer to allocate locks to shared variables in order to preserve atomic sections and minimize the number of locks used. The only similarity between this work and ours is the use of a pseudo-Boolean solver. The constraint problem formulation and the mostly Horn-SAT structure of our problems are novel and specific to our problem setting.

A number of patterns for declassifying data in code have been proposed by the literature. Some suggestions have been: building "helper" classes to declassify large data structures at once [3], using special `Closure` objects that require proofs of authority to declassify data [11, 5], only declassifying on method return [25], and placing declassifiers in special functions that are visible and controllable by the policy [12]. Each of these patterns can be captured by a particular type of hard constraint that can be applied during the generation of candidate declassifications. For future work, we will build a generalized framework that allows for the user to introduce new hard and soft constraints during the declassification proposal phase.

Existing work on automatically resolving errors in information-flow secure languages has focused on removing potential timing leak from code [1, 18]. We aim to eliminate a larger class of security errors, though we have not address automatic *removal* of illegal information flows. We have restricted our attention to placing declassifiers in programs as a way to summarize the illegal information-flows that an application enables. These declassifiers can be reviewed to give the programmer a better idea of how the application handles its secret data, but any legitimate security errors must still be removed manually.

8 Conclusion

In this paper we have presented a method for automatically placing declassifiers to allow the leakage of certain data in information-flow secure systems. Our experience has found that this approach is tractable and roughly proportional to the number of candidate declassifiers proposed by our system (column 4 of Table 1). In each case our tool selects a small fraction (less than 1%) of the number of potential declassification sites that would otherwise have to be

inspected manually, greatly reducing programmer burden.

References

[1] AGAT, J. Transforming out timing leaks. In *POPL* (New York, NY, USA, 2000), ACM Press, pp. 40–53.

[2] ALOUL, F. A., RAMANI, A., SAKALLAH, K. A., AND MARKOV, I. L. Solution and optimization of systems of pseudo-boolean constraints. *IEEE Trans. Comput. 56*, 10 (2007), 1415–1424.

[3] ASKAROV, A., AND SABELFELD, A. Secure implementation of cryptographic protocols: A case study of mutual distrust. In *ESORICS* (Milan, Italy, September 2005), LNCS, Springer-Verlag.

[4] BIEBER, P., CAZIN, J., MAROUANI, A. E., GIRARD, P., LANET, J.-L., WIELS, V., AND ZANON, G. The PACAP prototype: A tool for detecting Java Card illegal flow. *Java Card Workshop* (2000), 25–37.

[5] CHONG, S., VIKRAM, K., AND MYERS, A. C. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX Security* (2007).

[6] COUSOT, P., AND COUSOT, R. Galois connection based abstract interpretations for strictness analysis. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications* (28 June – 2 July 1993), Lecture Notes in Computer Science 735, pp. 98–127.

[7] DENNING, D. E., AND DENNING, P. J. Certification of programs for secure information flow. *Commun. ACM 20*, 7 (1977), 504–513.

[8] DILLIG, I., DILLIG, T., AND AIKEN, A. Static error detection using semantic inconsistency inference. In *PLDI* (2007), ACM, pp. 435–445.

[9] EEN, N., AND SORENSSON, N. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation 2* (2006), 1–26.

[10] EMMI, M., FISCHER, J. S., JHALA, R., AND MAJUMDAR, R. Lock allocation. In *POPL* (2007), M. Hofmann and M. Felleisen, Eds., ACM, pp. 291–296.

[11] HICKS, B., AHMADIZADEH, K., AND MCDANIEL, P. Understanding practical application development in security-typed languages. In *ACSAC* (Miami, Fl, December 2006).

[12] HICKS, B., KING, D., MCDANIEL, P., AND HICKS, M. Trusted declassification: High-level policy for a security-typed language. In *PLAS* (Ottawa, Canada, June 10 2006), ACM Press.

[13] JAJODIA, S., AND MEADOWS, C. *Inference problems in multilevel secure database management systems*. IEEE Computer Society Press, Los Alamitos, California, USA, 1995.

[14] KING, D., JAEGER, T., JHA, S., AND SESHIA, S. A. Effective blame for information-flow violations. NAS-TR-0069-2007, September 2007.

[15] LARRABEE, T. Test pattern generation using boolean satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems 11*, 1 (1992), 4–15.

[16] MALACARIA, P. Assessing security threats of looping constructs. In *POPL '07* (2007), ACM Press, pp. 225–235.

[17] MALACARIA, P., AND CHEN, H. Quantitative analysis of leakage for multi-threaded programs. In *PLAS '07*.

[18] MANTEL, H., AND KÖPF, B. Transformational typing and unification for automatically correcting insecure programs. *International Journal of Information Security (IJIS)* (2007).

[19] MYERS, A. C. JFlow: Practical mostly-static information flow control. In *POPL* (January 1999), pp. 228–241.

[20] MYERS, A. C., NYSTROM, N., ZHENG, L., AND ZDANCEWIC, S. Jif: Java + information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001.

[21] POTTIER, F., AND SIMONET, V. Information flow inference for ML. In *POPL* (New York, NY, USA, 2002), ACM Press, pp. 319–330.

[22] REHOF, J., AND MOGENSEN, T. A. Tractable constraints in finite semilattices. *Science of Computer Programming 35*, 2–3 (1999), 191–221.

[23] SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21*, 1 (Jan. 2003), 5–19.

[24] SABELFELD, A., AND SANDS, D. Declassification: Dimensions and principles. *Journal of Computer Security* (2007, to appear).

[25] SMITH, S. F., AND THOBER, M. Refactoring programs to secure information flows. In *PLAS* (2006), ACM Press, pp. 75–84.

[26] SOLAR-LEZAMA, A., TANCAU, L., BODÍK, R., SESHIA, S. A., AND SARASWAT, V. A. Combinatorial sketching for finite programs. In *ASPLOS* (2006), pp. 404–415.

[27] VAUGH, J. A., AND ZDANCEWIC, S. A cryptographic decentralized label model. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2007).

[28] XIE, Y., AND AIKEN, A. Saturn: A scalable framework for error detection using boolean satisfiability. *TOPLAS 29*, 3 (2007).