Towards Static Flow-based Declassification for Legacy and Untrusted Programs

Bruno P. S. Rocha*, Sruthi Bandhakavi†, Jerry den Hartog*, William H. Winsborough‡ and Sandro Etalle*,§

* Eindhoven University of Technology, The Netherlands
{b.p.s.rocha, j.d.hartog, s.etalle}@tue.nl
† University of Illinois at Urbana Champaign
sbandha2@illinois.edu
‡ University of Texas at San Antonio
wwinsborough@acm.org
§ University of Twente, Enschede, The Netherlands
sandro.etalle@utwente.nl

***Abstract*—Simple non-interference is too restrictive for specifying and enforcing information flow policies in most programs. Exceptions to non-interference are provided using declassification policies. Several approaches for enforcing declassification have been proposed in the literature. In most of these approaches, the declassification policies are embedded in the program itself or heavily tied to the variables in the program being analyzed, thereby providing little separation between the code and the policy. Consequently, the previous approaches essentially require that the code be trusted, since to trust that the correct policy is being enforced, we need to trust the source code.**

In this paper, we propose a novel framework in which declassification policies are related to the source code being analyzed via its I/O channels. The framework supports many of the of declassification policies identified in the literature. Based on flow-based static analysis, it represents a first step towards a new approach that can be applied to untrusted and legacy source code to automatically verify that the analyzed program complies with the specified declassification policies. The analysis works by constructing a conservative approximation of expressions over input channel values that could be output by the program, and by determining whether all such expressions satisfy the declassification requirements stated in the policy. We introduce a representation of such expressions that resembles tree automata. We prove that if a program is considered safe according to our analysis then it satisfies a property we call Policy Controlled Release, which formalizes information-flow correctness according to our notion of declassification policy. We demonstrate, through examples, that our approach works for several interesting and useful declassification policies, including one involving declassification of the average of several confidential values.

***Index Terms*—Security; Languages; Software verification and validation; Data flow analysis;**

I. Introduction

Programs dealing with sensitive data must prevent confidential information from flowing to unauthorized entities [1]. The classical security property of programs, *non-interference* [2], requires that publicly observable behavior is entirely independent of secret, secure input values. Type-based [3], [4], [5] and dataflow-based [6], [7], [8], [9] approaches have been proposed to statically analyze whether a given program enforces non-interference. In both approaches, each program variable is *labeled* with a security level (e.g., *high* for secret or *low* for public, though any lattice of labels can be supported). In type-based approaches, typing rules are defined such that if the program type-checks, the program is non-interferent. In dataflow-based approaches, an analysis calculates dependence relationships between program variables; non-interference is ensured if low variables are independent from high variables.

In general, non-interference is excessively restrictive: many programs that meet their security objectives fail to satisfy it. A classical example of this is that of a company policy that requires individual employee salaries be kept secret, but allows the average salary to be disclosed. Since non-interference prohibits any direct or indirect flow of secret information to a public output channel, *any* program that publishes the average salary violates it. To mitigate the rigidity of non-interference, one can explicitly allow exceptions to it in the form of *declassification policies* (see e.g., [10]), which identify circumstances under which information that depends on high-security inputs is permitted to flow to low outputs.

In type-based approaches, the exceptions to the standard flow are usually associated with specific points in the code. The programmer can specify the declassification policy by using a special *declass* command, which usually releases the information conditionally, depending on the value of a given expression over program variables. In frameworks of this kind, declassification policies are specified in a manner that is intimately tied to the program itself.

A serious drawback of this approach is that only someone with a deep understanding of the program can reliably write declassification policies for it. Everyone else is forced to trust blindly that the policies meet the required security objectives. When code is written by trusted programmers, this assumption may be acceptable, though even then it would be preferable to separate concerns and make the specification, maintenance, and review of declassification policies independent from the program. In the case of untrusted code, this arrangement is clearly unacceptable. Operators of systems that rely on such a program obtain little assurance that the declassification policies defined in it are appropriate. As pointed out by Zdancewic [11], one of the reasons why language-based techniques have not yet been widely adopted is that the enforcement approaches require the programmer to worry not only about the correctness of the program logic, but also about

how to annotate the program so that it can be deemed secure.

This state of affairs implies that declassification policies cannot readily be applied to legacy code. Unless the legacy program satisfies strict non-interference—which, is uncommon—the only way to determine whether such programs satisfy information-flow objectives is through the laborious process of understanding the program well enough to design a program-specific declassification policy.

Recently, Banerjee et al. [12] have partly addressed the problem of separation of concerns. They introduce a form of declassification policies called *flowspecs*, which are specified separately from the program and instantiated at particular program points. Flowspecs are a combination of ordinary program assertions extended with agreement predicates, both of which refer to local and global program variables. Although, this approach goes a long way toward separating the policy from the code, the policies still require an intimate knowledge of the program variables. As a result, program assertions are heavily tied to the programs being analyzed. Additionally, since their analysis uses the flow-insensitive, type-based approach, they require that programs disallow assigning new values to high variables prior to their use in expressions to be declassified. This means that programs need to be written in a policy-specific manner for them to be deemed valid, which is at odds with the application of their approach to legacy code.

To stress this point further, Hicks, et al. [13] conclude that although Jif is the most advanced security typed programming language, it is not ready for mainstream use because it requires considerably more programmer effort to write a working program than in a conventional language. In light of this observation, we believe there is need for an information flow analysis framework that does not require programmer annotations and which considers programs and policies as independent entities. This would result in greatly reducing the effort required to program an application.

Contribution: In this paper we introduce a novel approach for the specification and the (static) verification and enforcement of declassification policies that are independent of the code to which they are applied. The novelty of our approach lies in the combination of the following features: (a) it supports user-defined declassification policies, (b) code and policy are separated and independent from each other, (c) it allows one to analyse and apply declassification policies to unannotated and untrusted code.

We believe that this work can be seen as a novel general *methodology* within which verifiable analyses can be constructed that determine whether untrusted and legacy code enforces such code-independent policies, together with a particular *application* of the methodology implementing a particular analysis for determining whether a particular graph-based form of declassification policy is enforced by the input program. The methodology provides a basis for further work on even more expressive representations.

Rather than referring to particular program commands, our policies identify (sets of) expressions over values obtained from secret input channels; the values of these expressions are thus identified as candidates for declassification. Our approach to program analysis deems a program to be secure if it is able to determine that public output values depend on secret inputs only via expressions thus identified. Consequently, programs can be written without awareness of the formal declassification policies or of how the analyzer works, and no special command is used in programs to specify declassification.

Our declassification policies use *graphs* to represent sets of expressions over values obtained from input channels. This allows us to express and to deal efficiently with declassification policies that refer to iterative constructs such as loops (as in the example in which the average salary may be disclosed and the individual wages must remain secret). In present approaches, to declassify the result of a looping program using standard flow-based techniques, one is required to manually introduce simplifications, which often consist of determining the fix-points of loops. On the other hand, type-based techniques usually rely on the programmer to identify in the code iterative declassification expressions.

Our declassification policies represent values that are permitted to be made public. Expressions that may be computed by the program under analysis are also represented by a form of an expression graph that incorporates representations of variables and I/O channels, and captures the dependencies (control and data) of output expressions on values obtained from input channels. We augment the power of our expression graphs to allow them to express the (non-regular) property that values obtained from input channels are given by distinct read operations, thus enabling our policies to require, for instance, that an expression representing the average of input values must refer to multiple distinct values read from the input channel, and not multiple references to the value returned by a single read operation. A graph matching mechanism is used to ensure that the expressions are declassifiable per the policy.

Technically, our principal contribution is the introduction of a form of Conditioned Gradual Release (CGR) called Policy Controlled Release (PCR) —a more flexible security property that replaces non-interference—and a result that shows this property is satisfied by programs deemed valid by our analysis[1]. Compared to the prior definition of CGR [12], ours is much simpler and more intuitive because it can be expressed purely on the observable behaviour of programs rather than needing details on program executions.

We believe that our work takes a first step in a new direction for the information-flow field. The analysis of legacy and untrusted programs, along with a program-independent, policy-based declassification mechanism represents an important step towards bridging the gap between academic research in the field and its widespread adoption in industry. Due to the novelty of the approach, we build our mechanism over some simple assumptions: we use a simple imperative toy language; we define the matching mechanism mathematically, leaving

[1]While PCR is termination-sensitive, our analysis and theorem are termination-insensitive in the sense that our analysis may deem valid a program that leaks secret information by failing to terminate during a while loop that is controlled by a nondeclassifiable expression.

specification and analysis of algorithms out of the scope; and we leave some operational issues untreated (but discussed). However, we pave the way for these assumptions to be relaxed, towards a mechanism that will be able to analyze legacy systems using newly created declassification policies.

The rest of the paper is organized as follows: Section II presents detailed examples that illustrate our approach. In Section III we define our language syntax and semantics, as well as the program expression graphs and declassification policies, and one more example. Our matching mechanism is presented in Section IV. The security property is discussed in Section V, and the soundness of our analysis is shown in Section VI. Finally, Section VII presents related work, and Section VIII contains our discussion and conclusions of this work.

II. Example

In this section we illustrate by means of two examples the mechanism of our framework. The first example refers to one of the classical situations requiring declassification: authentication and password matching. The basic security requirement is that user information should not flow to the output channel, with one exception (captured by the declassification policy): boolean queries on the user's record may be declassified. Now, in order to authenticate the user, 3 methods are possible. If the user's record is "complete" and the user has a given credential, the function *validate* can check this credential. This is the preferred method for authentication. If, however, the user does not have the required credential, but his record is complete, then the same *validate* function can be applied over the user's last name, validating the user's name against a list. Finally, if the user's record is not complete, then the system prompts for a password, from another input channel, and use function *verify* to check it along with the user login name. In the end, the result of the authentication is sent to the output channel. The example program is given below. The language it uses is a standard imperative programming language, with no special security constructs, which will be used throughout the paper. The inputs and outputs to the program are specified using input and output channels and represented with Greek letters, as further explained in Section III. Channel α returns the record with the user information, channel β is used to retrieve a password from the user, if necessary, and channel γ is the output channel to where authorization information is sent.

Example 1. *Authentication program:*

```
struct x := α;
string f;
bool v;

if iscomplete(x) then
    if hascred(x) then
        f := credential(x);
    else
        f := lastname(x);
    v := validate(f);
else
    f := login(x);
    string y := β;
    v := verify(f, y);
γ := v;
```

Pre-processing and conversion to SSA: Our analysis works on code that has already been pre-processed in the following way: (1) operators are translated into functions (e.g., $a + b$ becomes $add(a, b)$), (2) only one function is allowed per assignment, i.e., assignments of complex expressions are broken into several assignments, (3) conditions on control-flow commands (`if` and `while`) refer to a single boolean variable.

We also convert a program into the Static Single Assignment (SSA) format using standard methods [14]. SSA is a known intermediate representation form for programs, in which every variable is assigned exactly once. Variables being assigned more than once are renamed (with a different name for each assignment: typically the original name with a subscript). For variables that are modified in the body of branching statements (e.g. conditionals and loops), the translation algorithm generates a new variable name at the *join points* (at the end of the conditional or the loop). Moreover, a new function ϕ is introduced, which takes as input the variable values from all the branches, and outputs the value from the branch that was taken. During the translation, we additionally annotate the ϕ function with the conditional variable of the branch to which the ϕ function is associated. The technique for computing SSA form of a program has been proved to be tractable. For more information on it refer to [14], [15].

Example 2. *Authentication program in SSA format:*

```
struct x1 := α;
bool c1 := iscomplete(x1);
string f0;
bool v0;

depends(β, c1);
if c1 then
    c2 := hascred(x1);
    if c2 then
        f1 := credential(x1);
    else
        f2 := lastname(x1);
    f3 := φ_c2(f1, f2);
    v1 := validate(f3);
else
    f4 := login(x1);
    string y1 := β;
    v2 := verify(f4, y1);
v3 := φ_c1(v1, v2);
f5 := φ_c1(f3, f4);
γ := v3;
```

Note that the conditions are syntactically associated with the ϕ-functions. Also, the *depends* command is generated

during the pre-processing and serves the purpose of making the control dependence between channel β and variable c_1 explicit, since the input occurs inside the conditional. This will be further explained in the next section.

Expression Graph: An expression graph is an abstraction for representing the set of expressions that may be assigned to a variable (or to more variables), taking into consideration the input channels and the constants that a program refers to. In an expression graph nodes represent variables, constants and I/O channels, whereas directed edges represent assignments. The labels on the edges denote the functions used in the assignments, while the subscripts indicate the indices of the arguments from the parent nodes. Edges of ϕ-functions are dashes as they are used to represent distinct paths that information can follow during an execution, each path separately creating a set of expressions. The `control` edge illustrates that there is a control dependency between two nodes, the parent being the variable representing the control expression. Figure 1 shows the expression graph g associated with the variable v_3 of our program. For clarity, a control edge between c_1 and β is omitted, since c_1 also causes a control dependency in v_3, and it will be analyzed anyway.

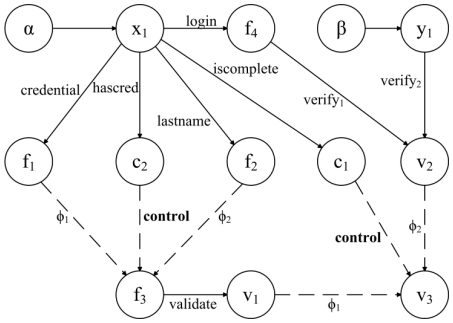

Fig. 1. Expression graph for variable v_3 of authentication program

Policy Graph: Declassification policies are also represented using graphs. In fact, a policy graph is similar to the expression graphs associated with program variables, except for some key differences, including: (1) nodes can be labeled with "wildcards", i.e., labels in the form $*$, (2) certain nodes in the policy are marked as "final nodes" (represented by the double lined circle), representing expressions that can be declassified. A declassification policy consists of a graph which might contain several disjoint components (to allow multiple expressions to be released). The policy graph, d, for our authentication program in Example 2 is given in Figure 2. We know that information from either channels α and β cannot directly flow to the channel γ, the policy of Figure 2 allows such a flow under a few additional conditions. The following operations are allowed: two boolean checks on the user's record α (if it has a credential and if is a complete record), two validation operations over user's information (validation through the credential or the last name), and a verification of the user's login against a supplied password from channel β. The final nodes $*_1$, $*_2$, $*_3$, $*_5$ and $*_7$ represent the expressions that can be declassified.

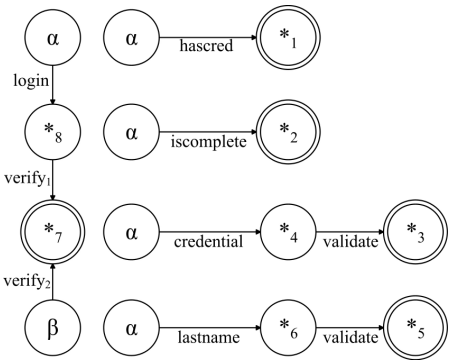

Fig. 2. Policy graph for example of authentication program

Policy Matching: Now that we have both the program and the policy graph, we can check if the program is *safe*. In our program the (low) output γ is assigned the value of variable v_3, so what we now have to check is that the paths in the program graph indicating the flow of information from a high input to v_3 are safe, i.e., that they match at least one component of the declassification policy. This analysis is done in two stages: first all data dependencies of a node are checked, later in the second stage the control dependencies are checked. The node representing v_3 in the graph has 3 *information paths* (defined in the next sections, not the standard concept of a path in a graph) reaching it: (a) one that comes from channel α passing through nodes x_1, f_1, f_3 and v_1, (b) another also coming from α, but passing through f_2 instead of f_1, and the final one (c) coming from both α and β, converging on node v_2. These paths represent the three possible outcomes of the nested `if` commands.

To determine that the node v_3 is safe we will first analyse its parents. First, node v_2, has only information path (c) reaching it. This path matches the leftmost component of our declassification policy in Figure 2, and we say that node v_2 *simulates* node $*_7$, meaning that all expressions possibly held by v_2 are recognized by $*_7$. Or, in other words, $v_2 \sim_{g,d} *_7$. Because of this, this path to v_2 is marked as *data dependency safe*, and so is the node, since this is its only path reaching it (note that the graphs are not exactly the same, our definition of policy simulation handles this properly). Since v_2 has no additional control dependencies (the dependency with c_1 is treated for v_3), we now know it is a *safe* node.

With (node) v_2 being safe, we now analyse v_1. This node has two information paths (a) and (b) reaching it. We can see that, for each path, v_1 simulates a final node of each of the 3-node components of the policy ($*_3$ and $*_5$), on the bottom of Figure 2. Thus, both paths are data dependency safe, and so is the node itself. There is however a control dependency that we have to consider, with c_2 that reaches v_1. But here node c_2 simulates the final node of the topmost policy component ($*_1$), thus making it safe (it has no control dependencies or other paths) and thus making v_1 *control dependency safe*. Therefore, we now know that v_1 is a safe node.

We can now go back to v_3. Since its two parent nodes are

safe, we know that v_3 is a data dependency safe node, since all the paths were covered. In order to demonstrate it is also control dependency safe, we need to show that node c_1 is safe, this is done by showing that the node simulates a final node of a policy graph ($*_2$). Thus, v_3 is a safe variable and the program's expression graph is deemed valid.

Second Example: We now provide a second example, that will be referred to throughout the paper. This example involves a policy which allows the declassification of expressions in a given recursive pattern, represented in the code by a looping structure. For this, this example uses control context annotations on the edges of the program expression graphs. These annotations were omitted on the previous example, for clarity. This example is inspired by another classical need for declassification: statistical calculations on secure data (where high data should not be released but statistics on it may be declassified). The program, given below (already pre-processed), calculates the average of the entries in a given data structure. Channel α returns the next element of a sequence of salaries of an organization. The code below fetches all the salaries from the structure, calculates their average, and then sends the result to output channel γ.

Example 3. *Average calculation program in SSA format:*

```
int a1 := 0;
int i1 := 0;
int l1 := length(α);
bool c1 := leq(i1, l1);

while (c3 := φc3(c1, c2);
       a3 := φc3(a1, a2);
       i3 := φc3(i1, i2);
       c3) do
    int t1 := α;
    a2 := add(a3, t1);
    i2 := add(i3, 1);
    c2 := leq(i2, l1);

a4 := div(a3, l1);
γ := a4;
```

Note that the ϕ-functions are placed along with the loop condition and the program semantics would require that the ϕ assignment be executed even if the loop is not taken, but also once after each iteration [16]. Again, the output channel γ receives the value of a_4, so we need to prove that a_4 is safe. To do so, we produce the expression graph associated to it (Figure 3). For the sake of clarity, Figure 3 only includes data-dependencies of a_4. Since we also have to consider the control dependencies, and the only control dependency of a_4 goes through c_3, we represent the graph associated with node c_3 separately in Figure 4. The numeric annotations on the edges indicate control contexts in which assignments are performed. With that, the assignments that happen within the loop have its corresponding edges marked with 1. The other edges are part of control context 0, with their annotations omitted.

The policy graph for the average example is given in Figure 5. This policy allows for the release of a sequence of additions over entries from input α. The final node $*_3$ represents the sum expression.

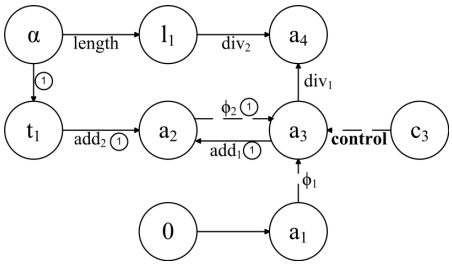

Fig. 3. Expression graph for variable a_4

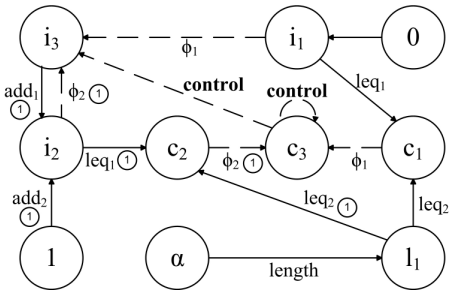

Fig. 4. Expression graph for variable c_3

The policy contains an additional constraint that states that no individual α-values should reach $*_3$ more than once (every access to α must be unique). Assuming d is the policy graph, we say that $(\alpha, *_3) \in uni(d)$. This is called an *input uniqueness relation*, we discuss how to express this in the next section.

We also assume that there is an omitted component of the policy graph that specifies that the expression $length(\alpha)$ can be declassified.

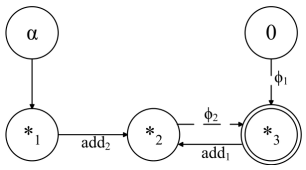

Fig. 5. Declassification policy graph for average example

This example program is deemed valid by the policy. Node a_3 simulates node $*_3$ on the policy and the α-uniqueness constraint is satisfied through the use of the control context annotations. Variable l_1 holds $length(\alpha)$, which is also authorized as previously mentioned. This makes c_3 being marked as safe, which in turn makes a_3 control dependency safe. Therefore, a_4 is marked as safe. The mechanisms used in this process are detailed in the next sections.

It is important to note that, since our approach works as a static analyzer, it is beyond the focus of our representation mechanism (i.e. graphs) to represent the run-time behaviour of the program, including the number of times a given loop

runs. This problem, however, can be treated by a combination of static analysis and runtime enforcement, discussed in Section VIII, but out of the scope of this paper, aimed at static analysis only.

III. Language: Syntax and Semantics

In this section we introduce the syntax and semantics of our language, we present programs' expression graphs and how they are created, together with a soundness theorem. Finally, we present the declassification policies.

Program Syntax and Semantics: **Var** is a set of variables, x, y, z, c, range over **Var** and may have subscripts; c is usually a boolean variable. Additional **IO** variables (**IO** $=$ **In** $\cup$ **Out**) represent input/output channels. We use α, β to denote input channels, γ, δ to denote output channels, and θ to range over all of **IO**. We additionally use ρ to range over **Var**$+$**IO**. Input channels are regarded as streams of values and are indexed to indicate specific input values; e.g., α_n denotes the n-th input value of input channel α.

Functions are defined the usual way. Constants are functions of arity 0, and we use N to denote them. Expressions are obtained by combining functions, variables (also **IO**) and constants in the usual way.

We use a simple imperative language with assignment, conditionals and loops, already translated to SSA form. To simplify the presentation, we assume that all operators are applied using prefix notation (e.g., writing $add(a, b)$ instead of $a + b$), with at most one function per assignment (no nesting); also, expressions on conditionals refer to a single boolean variable. Any program can be translated to this format in a straightforward manner. Regarding the SSA translation, ϕ-functions always have the form $x := \phi_c(a, b)$, where c is the conditional variable that generated that ϕ-function. In `while` expressions, $\overline{C}$ represents the ϕ-functions added by the SSA translation, which are evaluated once if the loop is not taken, and at every iteration otherwise.

Definition 4. *A program $C \in Prog$ is defined by the following syntax:*

$$\begin{aligned} C ::= {} & \texttt{skip} \mid x := \alpha \mid \gamma := x \mid x := f(y_1, \ldots, y_k) \\ & \mid x := \phi_c(a, b) \mid \mathit{depends}(\theta, c) \mid C_1 \,;\, C_2 \\ & \mid \texttt{if } c \texttt{ then } C_1 \texttt{ else } C_2 \mid \texttt{while } \overline{C} \,;\, c \texttt{ do } C \end{aligned}$$

The command $\mathit{depends}(\theta, c)$ is a special command that helps our non-standard semantics keep track of control dependence on I/O channels. It is added to the program during pre-processing: $\mathit{depends}(\theta, c)$ is inserted every time an input or output operation occurs inside a conditional, and relates the channel with the conditional under which it occurs. The command is added just before the conditional block in which the operation takes place.

Next, we define the program semantics, starting from the notion of state. Note that we present an instrumented semantics, in the sense that the state of the process keeps track of certain information useful for proving the compliance of our validation mechanism.

A *state* $\sigma \in \Sigma$ is a 4-tuple $\langle E, I, O, PC \rangle$, where

$$\begin{array}{rcrcl} E & \in & \mathcal{E} & = & \textbf{Var} \to \textbf{Exp}\langle \textbf{In} \times \mathbb{N} \rangle \\ I & \in & \mathcal{I} & = & \textbf{In} \to \mathbb{N} \\ O & \in & \mathcal{O} & = & \textbf{Out} \to \mathcal{P}(\textbf{Exp}\langle \textbf{In} \times \mathbb{N} \rangle) \\ PC & \in & \mathcal{PC} & = & (\textbf{Var} + \textbf{IO}) \to \mathcal{P}(\textbf{Exp}\langle \textbf{In} \times \mathbb{N} \rangle) \end{array}$$

E is a mapping from variables to expressions on indexed input channels, keeping track of the expression over the input that a variable holds; I is a mapping from input channels to numeric indexes, keeping track of the index of the next value to be read (so $I(\alpha)$ denotes index of the next value to be read from channel α); initially, $I(\alpha) = 1$ for every input channel α. O maps each output channel to the set of expressions (on indexed inputs) that could be sent over that channel. Finally, PC maps variables and channels (both input and output) to sets of expressions on indexed inputs, which records the implicit information flows, i.e. the expressions on which the variables and channels are conditionally dependent. Given a state σ, we write E_σ to indicate its first component, I_σ for the second, etc. We omit σ if it is clear from the context, thus e.g. $E(x)$ denotes the expression held by x in the "current" state.

Next, we define *environments* which provide the input to the program through the channels. We have a straightforward channel model where the channels are independent of each other. In Section VIII we discuss how to extend this to more intricate channel models.

$$\pi \in \Pi : \textbf{In} \times \mathbb{N} \to \textbf{Val}$$

Finally, we define a configuration over which the semantics are defined.

Definition 5 (Configuration). *A* configuration $\omega \in \Omega$ *is a triple $\langle C, \sigma, \pi \rangle$, where C is a program, σ a state and π an environment.*

Note that the environment determines the inputs that have been or will be provided to the program and (due to our channel model) does not change during the execution of the program. The operational semantics is presented in Figure 6. The transitions between configurations have a label ($\in$ **Obs**) representing what can be observed externally when that transition occurs; a τ label represents a non-observable transition. In our case, the only observable action is the output, showing the channel and the value being sent over the channel. I.e. $o \in$ **Obs** is τ or $out(\gamma, v)$ for some output channel γ and value v.

We use square brackets to denote substitution e.g. $E_\sigma[E_\sigma(y)/x]$ returns a new $E_{\sigma'}$, in which $E_{\sigma'}(x) = E_\sigma(y)$. For changes in the state, we only indicate the components for which σ' differs from σ. Our semantics treats ϕ-functions in a special way. Unlike the standard functions, ϕ-functions are evaluated as soon as they appear. Function EV makes this evaluation. According to standard definition of the ϕ-functions in SSA form, the function returns the variable that has been defined most recently. The boldface **f**, used in function V indicates that the function is actually evaluated to a value.

The *initial state* σ_{init} is the state in which no channels have been read yet ($I(\alpha) = 1$), all variables are undefined

$$\langle x := \alpha, \sigma, \pi\rangle \xrightarrow{\tau} \langle \texttt{skip}, \sigma', \pi\rangle \quad \textit{(Input)}$$
$$\text{where} \quad E_{\sigma'} = E_\sigma[\alpha_{I_\sigma(\alpha)}/x] \quad I_{\sigma'} = I_\sigma[I_\sigma(\alpha)+1/\alpha] \quad PC_{\sigma'} = PC_\sigma[PC_\sigma(\alpha)/x]$$

$$\langle \gamma := x, \sigma, \pi\rangle \xrightarrow{o} \langle \texttt{skip}, \sigma', \pi\rangle \quad \textit{(Output)}$$
$$\text{where} \quad o = out(\gamma, V(E_\sigma(x), \pi)) \quad O_{\sigma'} = O_\sigma[O_\sigma(\gamma) \cup E_\sigma(x)/\gamma] \quad PC_{\sigma'} = PC_\sigma[PC_\sigma(\gamma) \cup PC_\sigma(x)/\gamma]$$

$$\langle x := f(y_1, \ldots, y_k), \sigma, \pi\rangle \xrightarrow{\tau} \langle \texttt{skip}, \sigma', \pi\rangle \quad \textit{(Assign)}$$
$$\text{where} \quad E_{\sigma'} = E_\sigma[f(E_\sigma(y_1), \ldots, E_\sigma(y_k))/x] \quad PC_{\sigma'} = PC_\sigma[PC_\sigma(y_1) \cup \ldots \cup PC_\sigma(y_k)/x]$$

$$\langle x := \phi_c(a,b), \sigma, \pi\rangle \xrightarrow{\tau} \langle \texttt{skip}, \sigma', \pi\rangle \quad \textit{(Phi)}$$
$$\text{where} \quad E_{\sigma'} = E_\sigma[EV(\phi_c(a,b), \sigma)/x] \quad PC_{\sigma'} = PC_\sigma[E_\sigma(c) \cup PC_\sigma(c) \cup PC_\sigma(a) \cup PC_\sigma(b)/x]$$

$$\langle \textit{depends}(\theta, c), \sigma, \pi\rangle \xrightarrow{\tau} \langle \texttt{skip}, \sigma', \pi\rangle \quad \textit{(Depends)}$$
$$\text{where} \quad PC_{\sigma'} = PC_\sigma[PC_\sigma(\theta) \cup E_\sigma(c) \cup PC_\sigma(c)/\theta]$$

$$\langle \texttt{if } c \texttt{ then } C \texttt{ else } \overline{C}, \sigma, \pi\rangle \xrightarrow{\tau} \langle C, \sigma, \pi\rangle \text{ if } V(E(c), \pi) = \texttt{true} \quad \textit{(If 1)}$$
$$\xrightarrow{\tau} \langle \overline{C}, \sigma, \pi\rangle \text{ if } V(E(c), \pi) = \texttt{false} \quad \textit{(If 2)}$$

$$\langle \texttt{while } \overline{C} \texttt{ ; } c \texttt{ do } C, \sigma, \pi\rangle \xrightarrow{\tau} \langle \overline{C}, \sigma, \pi\rangle \text{ if } V(E(c), \pi) = \texttt{false} \quad \textit{(While 1)}$$
$$\xrightarrow{\tau} \langle \overline{C} \texttt{ ; } C \texttt{ ; while } \overline{C} \texttt{ ; } c \texttt{ do } C, \sigma, \pi\rangle \text{ if } V(E(c), \pi) = \texttt{true} \quad \textit{(While 2)}$$

$$\langle \texttt{skip ; } C, \sigma, \pi\rangle \xrightarrow{\tau} \langle C, \sigma, \pi\rangle \quad \textit{(Skip)}$$

$$\frac{\langle C, \sigma, \pi\rangle \xrightarrow{o} \langle C', \sigma', \pi\rangle}{\langle C \texttt{ ; } \overline{C}, \sigma, \pi\rangle \xrightarrow{o} \langle C' \texttt{ ; } \overline{C}, \sigma', \pi\rangle} \quad \textit{(Seq)}$$

$$EV : \mathbf{Exp}\langle\mathbf{Var}\rangle \times \Sigma \to \mathbf{Exp}\langle\mathbf{In} \times \mathbb{N}\rangle$$
$$EV(\phi_c(a,b), \sigma) = \begin{cases} E(a) & \text{if } a \text{ has been most recently defined;} \\ E(b) & \text{if } b \text{ has been most recently defined.} \end{cases}$$

$$V : \mathbf{Exp}\langle\mathbf{In} \times \mathbb{N}\rangle \times \Pi \to \mathbf{Val}$$
$$V(e, \pi) = \begin{cases} \pi(\alpha, n) & \text{if } e = \alpha_n; \\ \mathbf{f}(V(e_1, \pi), \ldots, V(e_n, \pi)) & \text{if } e = f(e_1, \ldots, e_n). \end{cases}$$

Fig. 6. Program semantics

($E(x) = \perp$) and no output has been written to any channel ($O(\gamma) = \emptyset$). A *run* of program C in environment π is a sequence of configurations, starting from the initial configuration and linked by transitions, i.e., $t \in (\mathbf{Obs} \times \Omega)^*$ in which for $t = \langle o_0, \omega_0\rangle.\langle o_1, \omega_1\rangle \ldots \langle o_n, \omega_n\rangle$, $o_0 = \tau$, $\omega_0 = \langle C, \sigma_{init}, \pi\rangle$, and for each i, such that $0 \leq i < n$, $\omega_i \xrightarrow{o_{i+1}} \omega_{i+1}$ is a transition given by the semantics (Figure 6). We say the run is a *full run* if no steps are possible from end state ω_n otherwise the run is called a *prerun*. We write $\iota(t)$ for the sequence of (visible) output actions taken in t and $t_1 \equiv_{\text{out}} t_2$ if $\iota(t_1) = \iota(t_2)$. For sets of traces T, T' we put $T \equiv_{\text{out}} T'$ if $\forall t \in T : \exists t' \in T' : t \equiv_{\text{out}} t'$ and vice versa. Finally, we also write $Run(C, \pi)$ for the runs of C (note that for each prerun in t there is exactly one run t' which extends t with one step).

Program Expression Graph: Now we define how a graph is built from the program. We consider a directed typed graph, defined as $G = (V, E)$ where $V \subseteq$ **Vertex** and $E \subseteq$ **Edge** are the sets of vertices and edges, respectively. A vertex n has the form (l, t), where l is the label and t is the type, which can be `var`, `in`, `out`, and `const`, for variables, I/O channels and constants, respectively. For convenience, n_l denotes the vertex with label l, and we assume that the type of the node is clear from the label, e.g., n_x, n_α, n_γ, n_ρ and n_N are nodes of types `var`, `in`, `out`, any and `const`, respectively. An edge e has the form (n, n', t, i), where n and n' are the origin and destination vertices, respectively; t is the edge type, which can be `plain` (for assignments with no function application), `control` (for control dependencies between boolean variables in conditionals and variables assigned inside the conditional block), or a function name f, for edges that represent function applications, and i is an index that represents the control context in which the assignment represented by the edge takes place. We use f_i when the source node of the edge is the i-th argument of function f.

To build the graph we use the function G, defined in Figure 7, which takes a command C and two control context indexes i and j as arguments and returns the corresponding graph. Argument i represents the "current" control context index, whereas j represents the highest index and it is used to guarantee that different control contexts have distinct indexes. We write $G(C)$ as a short to $G_{0,0}(C)$. The ϕ-functions generated on SSA translation are used to handle control flow dependencies. Note that there are multiple definitions for assignments, according to the format of the RHS operator. Also, for the control context index, ϕ-functions in loops receive a special treatment. In these cases, the function is called and returns the first argument (ϕ_1 edge) aways once, regardless if the loop runs or not, whereas it is called and returns the second argument (ϕ_2 edge) as many times as the loop runs. Thus, the ϕ_1 edge is labeled with the control context index of before entering the loop, and ϕ_2 is labeled with the same control context of the loop. This is represented by function $\overline{G}$. Also, function $mcc(g)$ takes a graph g and returns the highest control context index in it.

$$\begin{aligned}
G : Prog \times \mathbb{N} \times \mathbb{N} &\to \mathcal{G} \\
G_{i,j}(\texttt{skip}) &= \emptyset \\
G_{i,j}(C_1 \texttt{ ; } C_2) &= G_{i,j}(C_1) \cup G_{i,j'}(C_2) \text{ where } j' = mcc(G_{i,j}(C_1)) \\
G_{i,j}(\texttt{if } c \texttt{ then } C_1 \texttt{ else } C_2) &= G_{j+1,j+1}(C_1 \texttt{ ; } C_2) \\
G_{i,j}(\texttt{while } \overline{C} \texttt{ ; } c \texttt{ do } C) &= \overline{G}_{i,j+1}(\overline{C}) \cup G_{j+1,j+1}(C) \\
G_{i,j}(x := \alpha) &= n_\alpha \xrightarrow[i]{} n_x \\
G_{i,j}(\gamma := x) &= n_x \xrightarrow[i]{} n_\gamma \\
G_{i,j}(x := y) &= n_y \xrightarrow[i]{} n_x \\
G_{i,j}(x := f(y_1, \ldots, y_k)) &= n_{y_1} \xrightarrow[i]{f_1} n_x, \ldots, n_{y_k} \xrightarrow[i]{f_k} n_x \\
G_{i,j}(x := \phi_c(a,b)) &= n_a \xrightarrow[i]{\phi_1} n_x, n_b \xrightarrow[i]{\phi_2} n_x, n_c \xrightarrow{\texttt{control}} n_x \\
G_{i,j}(\textit{depends}(\theta, c)) &= n_c \xrightarrow{\texttt{control}} n_\theta \\
\overline{G}_{i,j}(C_1 \texttt{ ; } C_2) &= \overline{G}_{i,j}(C_1) \cup \overline{G}_{i,j}(C_2) \\
\overline{G}_{i,j}(x := \phi_c(a,b)) &= n_a \xrightarrow[i]{\phi_1} n_x, n_b \xrightarrow[j]{\phi_2} n_x, n_c \xrightarrow{\texttt{control}} n_x
\end{aligned}$$

Fig. 7. Graph building function

Definition 6 (Expression Graph). *The expression graph $g \in \mathcal{G}$ of a program C is given by $G(C)$.*

We use $nodes(g)$ and $edges(g)$ to denote the sets of vertices and edges of g, respectively. We use $n \xrightarrow[i]{t} n'$ to denote that there is an edge of type t and control context i from node n to node n'. When $t = \texttt{plain}$, we use just $n \to n'$ as

a shorthand and we omit i when its value is irrelevant. In a similar fashion, $n \xrightarrow{w}{}^{*} n'$ denotes that there is a path between nodes n and n', with w being the sequence of labels on this path, and $n \xrightarrow[i]{w}{}^{*} n'$ denotes that the whole path w has the same control context i. We also use $\xrightarrow{\phi}$ to denote either $\xrightarrow{\phi_1}$ or $\xrightarrow{\phi_2}$. Using a notation analogous to that of bisimulation [17], we call an edge a τ-edge if its type is either `plain` or ϕ. Finally, we write $\not\rightarrow n$ to denote that the indegree of n is zero, and function $type(n)$ returns the type of a node n.

For our first theorem, we first present the definition of input-uniqueness.

Definition 7 (**Input Uniqueness on Expressions**). *An expression e : $\mathbf{Exp}\langle\mathbf{In}\times\mathbb{N}\rangle$ is said to be α-unique if every occurrence of α represents a distinct access on that input channel, i.e. every α_i has a distinct index i.*

We now define a notion of α-uniqueness for graph nodes n that will be used below to express the requirement that expressions recognized by n be α-unique. Given a graph g, this notion is represented by a set of pairs $uni(g) \subseteq \mathbf{In} \times \mathbf{Vertex}$ and $(\alpha, n) \in uni(g)$ indicates that n is intended to represent only α-unique expressions. In policy graphs (see Definition 10 below), this set is given explicitly. For program graphs, we derive it according to the following definition. (We believe that this definition is somewhat conservative in the sense that it may not extract all α-uniqueness pairs that could be derived in some cases, but it serves us well, is simple, and admits efficient computation.)

Definition 8 (**Input Uniqueness on Nodes**). *Let n_x be a variable node in the program expression graph g, and α be an input channel. We say that n_x is α-unique,*

$$(\alpha, n_x) \in uni(g)$$

if each path w from n_α to n_x has a unique control context i_w (i.e., for each such path w there exists i_w such that $n_\alpha \xrightarrow[i_w]{w}{}^{*} n_x$).

We now define function exp : $\mathbf{Vertex} \times \mathcal{G} \rightarrow \mathcal{P}(\mathbf{Exp}\langle\mathbf{In} \times \mathbb{N}\rangle)$ which makes precise the set of expressions represented by each graph node. We write $exp_g(n)$ to denote the set of expressions represented by node n in graph g, and we omit g when it is clear from context. So that the following definition can be applied to policy graphs (see Definition 10 below) as well as to program graphs, it is defined over nodes labeled with wildcard (n_*), as well as over nodes labeled as they are in program graphs.

$$\begin{array}{rcll} exp(n_N) & = & \{N\} & \\ exp(n_*) & = & \text{Const} & \text{if } type(n_*) = \texttt{const} \\ exp(n_\alpha) & = & \{\alpha_i \mid i \in \mathbb{N}\} & \\ exp(n_*) & = & \mathbf{In} \times \mathbb{N} & \text{if } type(n_*) = \texttt{in} \\ exp(n_\gamma) & = & \bigcup_{n' \rightarrow n_\gamma} exp(n') & \end{array}$$

$$\begin{array}{rcl} exp(n_x) & = & \\ \Psi_{n_x}\left(\begin{array}{ll} \bigcup_{n' \xrightarrow{\tau} n_x} & exp(n') \\ \cup & \{f(e^1, \ldots, e^k) \mid \\ & \exists n^i \xrightarrow{f_i} n_x, e^i \in exp(n^i), i = 1..k\} \end{array}\right) & \\ \Psi_n(E) & = & \{e \in E \mid \forall \alpha \in \mathbf{In} : \\ & & (\alpha, n) \in uni(g) \Rightarrow e \text{ is } \alpha\text{-unique}\} \end{array}$$

where Const denotes the set of (syntactical) constants and Ψ_n is a filter used to deal with input uniqueness, removing expressions which don't satisfy α-uniqueness if the node n holds that property. Note that in the above definition exactly one of the subsets of $exp(n_x)$ will be non-empty as each node either has a single plain edge, two ϕ edges or k incoming function edges. A node of type `const` with a wildcard $*$ label holds any constant as its expressions and a node of type `in` with wildcard label matches any indexed input α_i.

We also define the function $cexp$: $\mathbf{Vertex} \times \mathcal{G} \rightarrow \mathcal{P}(\mathbf{Exp}\langle\mathbf{In} \times \mathbb{N}\rangle)$ that computes all conditional expressions the value held by a node can depend upon.

$$cexp(n) = \{cexp(n') \mid n' \xrightarrow{t} n\} \cup \{exp(n'') \mid n'' \xrightarrow{\texttt{control}} n\}$$

We can finally state our first result, which is about the soundness of the graph translation. The proof is omitted and is available in the technical report [18].

Theorem 9 (**Soundness of the graph translation**). *Given a program C_0, environment π_0, let t be a run in $Run(C_0, \pi_0)$, and $g = G(C_0)$. For any configuration $\langle C, \sigma, \pi\rangle \in t$ we have that σ satisfies:*

$$\begin{array}{ll} (P_E) & \forall x \in \mathbf{Var} : E(x) \text{ is defined } \Rightarrow \\ & \quad (x, \texttt{var}) \in nodes(g) \wedge E(x) \in exp(n_x) \\ (P_I) & \forall \alpha \in \mathbf{In} : I(\alpha) > 0 \Rightarrow \\ & \quad (\alpha, \texttt{in}) \in nodes(g) \\ (P_O) & \forall \gamma \in \mathbf{Out} : O(\gamma) \text{ is defined } \Rightarrow \\ & \quad (\gamma, \texttt{out}) \in nodes(g) \wedge O(\gamma) \subseteq exp(n_\gamma) \\ (P_{PC}) & \forall \rho \in \mathbf{Var} + \mathbf{IO} : PC(\rho) \text{ is defined } \Rightarrow \\ & \quad (\rho, type(\rho)) \in nodes(g) \wedge PC(\rho) \subseteq cexp(n_\rho) \end{array}$$

(P_E) states that each variable has a corresponding node, and that the expression of the variable is contained in the set of possible expressions held by that node; (P_I) states that for each input channel accessed in the process there exists a corresponding node in the graph; (P_O) states that for each output channel there exists a corresponding node, and that the set of expressions sent to that output in the process is a subset of the set of possible expressions held by that node; finally, (P_{PC}) states that for each variable (and I/O channel), the set of conditional expressions that the variable depends on is equal to that set for the corresponding node.

Policy Expression Graph: Policy graphs work in the same way as program graphs, with a few key differences: (1) one or more nodes are marked as "final nodes"; (2) nodes can have "wildcards" as label, in the form of $*_i$, meaning that they can match any other node, regardless of the label; (3) edges don't have control context labels; and (4) input uniqueness relations

are provided with the policy, working as constraints over the recognized expressions. These differences are justified by the fact that the program graph is calculated, in order to represent all possible expressions that can be held by variables in the program, whereas policy graphs are supplied, recognizing the set of expressions that can be declassified. For clarity, we write $*^t$ to denote the wildcard on a node of type t, and just $*$ when $t = \texttt{var}$. The matching process between the policy and program graph is defined in Section IV.

Definition 10 (**Declassification Policy**). *A declassification policy is a graph $d \in \mathcal{D}$, with possibly disjoint components, in the form $d = (V, E, V_f, U)$, where $V \subseteq$ **Vertex** is a set of vertices, $E \subseteq$ **Edge** is a set of edges, $V_f \subseteq V$ is a set of final vertices and $U \subseteq$ **In** $\times V$ is a set of input uniqueness relations.*

The final vertices hold the expressions allowed to be declassified. Thus, the set of expressions allowed by a policy graph is determined by $\bigcup_{n_f \in V_f} exp_d(n_f)$. Also, we use $uni(d)$ to return the set of input uniqueness relations from a policy d. Thus, for our working example of the average salary, we have that the policy of Figure 5 recognizes the set of expressions $\{0, add(0, \alpha_i), add(add(0, \alpha_i), \alpha_j), \ldots\}$, with all indices on α being distinct, as $(\alpha, *_3) \in uni(d)$. It is important to point that this work addresses the problem of enforcing declassification policies, rather than specifying them. However, it is fairly straightforward to derive a rule that translates the policy graph to/from some form of regular expressions (e.g. regular tree expressions).

We now present another example, showing the use of wildcards in the policy. It is another of the classical examples of declassification, this time in presence of encryption: we have data that is sensitive if unencrypted, but its encrypted version can be declassified. The code below is already pre-processed: the input channel α provides a sensitive plain text file, β represents a cryptographic key. Output channel γ represents a low output.

Example 11. *Encryption program:*

```
text x1 := α;
int k1 := β;
x2 := enc(x1, k1);
γ := x2;
```

For our example, we consider a policy that allows any input to be declassified, as long as it is encrypted with a specific key, using a specific function. Figure 8 shows the graphs for both the policy and the program. In this case, node $*_2^{\text{in}}$ in the policy matches node α in the graph, and it is clear that the content of variable x_2 can be made public, matching final node $*_1$.

As mentioned in our first example, we don't consider a declassification to be invertible. For this example, one may think that, after x_2 has been marked as safe, a decryption function could be used to retrieve the original α value to a new x_3 variable. However, since the decryption function would need the decryption key, the inheritance from x_2 by itself would not be sufficient for x_3 to be marked as safe. The edge from input β to the decryption function would also need to be validated and this validation would not happen, as there is no policy that allows it, therefore making x_3 insecure.

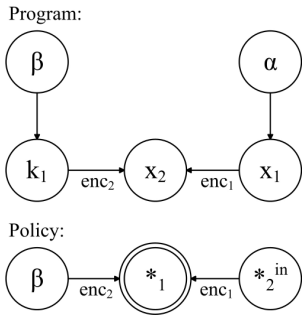

Fig. 8. Encryption program and its matching policy

IV. Policy Matching

Having defined the expression graphs of program and policy in the previous section we now introduce the mechanism that matches them. This will allow us to define which nodes are safe according to the policy. If all output nodes are safe, then the program represented by the graph is safe too. Note that multiple disjoint components of a declassification graph may be needed to show the safety of a program. To simplify this process we first extract the sub-graphs from the program's expression graph that could be validated separately (called *information paths*). Next we carry out the matching between an information path and a (single) component on a policy graph. It is important to point that in this section we present the definition of the matching mechanism, rather than an algorithm. Even though an algorithm can be directly derived from our definitions, due to space constraints we leave specification and analysis of such algorithm as future work.

An information path captures one way that expressions can flow into a node starting from input channels and constants. Multiple function edges to a same node represent the same expression held by that node, and all edges need to be included in the path. On the other hand, ϕ-edges represent points where control flow may branch, and therefore each ϕ-edge represents a distinct information path. Note that an information path may still have multiple incoming τ-edges because loops may cause us to reach the same node multiple times. We represent an information path by the set of edges it contains (the set of vertices for the graph can be obtained by collecting the source or destination of the edges). For a set S of information paths we use the notation:

$$\begin{aligned} S \oplus e &= \{g \cup \{e\} \mid g \in S\} \\ S \otimes S' &= \{g \cup g' \mid g \in S, g' \in S'\} \end{aligned}$$

The function ip calculates all the information paths that

reach node n, which is the set of sub-graphs satisfying:

$$\begin{array}{rcl} ip(n) & = & \{\emptyset\} \quad \text{if } \not\rightarrow n. \text{ Otherwise:} \\ ip(n) & = & \bigcup_{n' \xrightarrow{\tau} n} ip(n') \oplus (n', n, t) \quad \cup \\ & & \bigotimes_{n' \xrightarrow{f_i} n} ip(n') \oplus (n', n, t) \end{array}$$

If one information path is a sub-graph of another then validating the larger graph also validates the smaller so we only need to consider *maximal information paths*, i.e. maximal elements of ip.

$$mip(n) \quad = \quad \{g \mid g \in ip(n), \forall g' \in ip(n) : g \not\subset g'\}$$

The next step is to relate the maximal information paths to the policies. This is done by the notion of *policy simulation* which is a "bundled" weak simulation. We write $(n^i)_{i=1..k} \overset{f}{\Rightarrow} n'$ if $\exists n : \forall i \in \{1..k\} : n^i \ (\xrightarrow{\tau})^* \xrightarrow{f_i} (\xrightarrow{\tau})^* \ n'$. Also we call two nodes *similar* $n \simeq n'$ if they have the same type and either the labels are the same or one of them is a wildcard $*$.

Definition 12 (Policy Simulation). *A relation $\mathcal{R}$ between information path nodes and policy graph nodes is called a policy simulation if for all $(n, n_d) \in \mathcal{R}$ we have:*

$$\begin{array}{l} n \simeq n_d \wedge \\ (n' \xrightarrow{\tau} n \ \Rightarrow \ \exists n'_d : n'_d \ (\xrightarrow{\tau})^* \ n_d \wedge (n', n'_d) \in \mathcal{R}) \wedge \\ (\forall i \in \{1..k\} : n^i \xrightarrow{f_i} n \ \Rightarrow \ \exists n^1_d..n^k_d : (n^i_d)_{i=1..k} \overset{f}{\Rightarrow} n_d \ \wedge \\ \qquad\qquad\qquad\qquad \forall i \in \{1..k\} : (n^i, n^i_d) \in \mathcal{R}) \wedge \\ (\forall (\alpha, n_d) \in uni(d), \forall w : n_\alpha \xrightarrow{w}{}^* \ n \Rightarrow \exists i : n_\alpha \overset{w}{\underset{i}{\rightarrow}}{}^* \ n) \end{array}$$

We use $\sim_{g,d}$ to denote the largest policy simulation (i.e. the union of all of them) between information path g and policy graph d.

Next, we present a few supporting definitions for validating a program's expression graph. First, we define when a node n in an information path g is "safe" in terms of data dependencies. This is the case if it matches some final node of the declassification policy or all its parents are already safe; here $fnodes(d)$ returns the sets of final vertices on policy graph d.

$$\begin{array}{rcl} dds_g(n, d) & \equiv & (\exists n_f \in fnodes(d) : n \sim_{g,d} n_f) \vee \\ & & (\forall (\alpha, \texttt{in}) \in nodes(g), w : \alpha \xrightarrow{w}{}^* \ n \\ & & \Rightarrow \exists n' \in nodes(g) : \alpha \xrightarrow{w'}{}^* \ n' \xrightarrow{w''}{}^* \ n \wedge \\ & & dds_g(n', d)) \end{array}$$

Now we define a data dependency safe node, as a node in which all maximal information paths that reach it are data dependency safe. Function dds defines this relation.

$$dds(n, g, d) \quad \equiv \quad \forall p \in mip(n, g) : dds_p(n, d)$$

Similarly, a node is "control dependency safe" (CDS) if all nodes on which it has control dependencies, directly or indirectly, are DDS. For this definition, the whole graph is analyzed, instead of only individual paths. Relation cds captures the notion.

$$\begin{array}{rcl} cds(n, g, d) & \equiv & \forall n' \in nodes(g) : \exists n'' \in nodes(g) : \\ & & n' \xrightarrow{\texttt{control}} n'' \xrightarrow{w}{}^* \ n \Rightarrow dds(n', g, d) \end{array}$$

Then, a "safe" node is a node whose both data and control dependencies are safe, i.e. a node which is both DDS and CDS.

$$safe(n, g, d) \quad \equiv \quad dds(n, g, d) \wedge cds(n, g, d)$$

Finally, we can present the definition of a "valid" graph, which holds for a graph if all its outputs are safe.

Definition 13 (Graph Validity). *An expression graph g is marked as valid with respect to a policy d if the following is true:*

$$valid(g, d) \quad \equiv \quad \forall (\gamma, \texttt{out}) \in nodes(g) : safe(n_\gamma, g, d)$$

We say g is d-valid if valid(g,d).

With the matching mechanism defined, we can present its theorem of soundness. It states that if a node n in the graph simulates a node n_d in the policy graph, then the set of expressions possibly held by n is a subset of the set held by n_d in the policy. Once again, proof is omitted.

Theorem 14 (Soundness of the matching mechanism). *For a program's expression graph g and a policy d, the following relation holds:*

$$\begin{array}{c} \forall n \in nodes(g), n_d \in nodes(d) \ : \\ n \sim_{g,d} n_d \quad \Rightarrow \quad exp_g(n) \subseteq exp_d(n_d) \end{array}$$

For the next theorem, we define the notion of a "public" expression, in terms of a declassification policy. The relation is defined below.

$$\begin{array}{rcl} public(e, d) & \equiv & (\exists n_f \in fnodes(d) : e \in exp_d(n_f)) \vee \\ & & (e = f(e_1, \ldots, e_n) \wedge \\ & & public(e_1, d) \wedge \cdots \wedge public(e_n, d)) \end{array}$$

With this, we can present the theorem of safety between process and policy, demonstrating that if the corresponding graph of a program satisfies a policy, then the expressions on the process will also satisfy it. This theorem is a consequence of theorems 9 and 14.

Theorem 15 (Safety between process and policy). *For a program C_0, environment π_0 and t a run in $Run(C_0, \pi_0)$, any configuration $\langle C, \sigma, \pi \rangle \in t$, the graph $g = G(C_0)$, and a policy d, the following relations hold:*

$$\begin{array}{ll} (i) & \forall x \in \textbf{Var} : E_\sigma(x) \textit{ is defined} \wedge \\ & dds(n_x, g, d) \Rightarrow public(E_\sigma(x), d) \\ (ii) & \forall \gamma \in \textbf{Out} : O_\sigma(\gamma) \textit{ is defined} \wedge \\ & dds(n_\gamma, g, d) \Rightarrow \forall e \in O_\sigma(\gamma) : public(e, d) \\ (iii) & \forall \rho \in \textbf{Var} + \textbf{IO} : PC_\sigma(\rho) \textit{ is defined} \ \wedge \\ & cds(n_\rho, g, d) \Rightarrow \forall e \in PC_\sigma(\rho) : public(e, d) \end{array}$$

(i) states that if a variable in the program has its corresponding node in the graph (which is guaranteed to exist by Theorem 9) being data dependency safe, then the expression held by that variable in the process is safe (i.e. allowed by the policy); (ii) states that if an output channel in the program has its corresponding node in the graph being data dependency safe, then all expressions sent to it in the process are safe; finally, (iii) states that if a variable or I/O channel has a corresponding node in the graph being control dependency safe, then all conditional expressions of the variable (or I/O channel) in the process are safe.

V. Security Property

In this section we define our reference security property called *Policy Controlled Release*. It is an "end-to-end" property in the sense that it bounds the knowledge that an attacker can gain by observing information released on output channels during any collection of runs. Our property closely follows the Conditional Gradual Release (CGR) given by Banerjee et al. [12], though our variant differs from the original definition in several important respects, being simpler and independent of characteristics of the program's execution. CGR itself is a variant of the Gradual Release [19] property. To simplify the discussion, we assume that information obtained from all the input channels is confidential and can be modified only by the target machine (on which the program runs). Reading from an input channel is not visible to an outsider. On the other hand, any information placed on the output channels is regarded as public. Releasing information from the secret input channels to the public output channels is permitted only according to declassification policies. Recall that we have also assumed that the input channels are non-interactive in the sense that reading data from one input channel, has no effect on the values obtained from other input channels. We discuss the relaxation of these assumptions in Section VIII.

Two environments are said to be d-Equivalent if the values of the declassifiable expressions are the same in both the environments. Evaluating the expressions represented by final nodes in a policy (see V in Section III) gives the actual values that can be declassified.

Definition 16 (d**-Equivalent Environments** ($\approx_d$))**.** *Given a declassification policy d, two environments π_1 and π_2 are said to be* d-equivalent*, $\pi_1 \approx_d \pi_2$, if $\forall n_f \in fnodes(d)\,.\,\forall e \in exp_d(n_f)\,.\,V(e,\pi_1) = V(e,\pi_2)$.*

Lemma 17. *Given a declassification policy d, two environments π_1 and π_2, if $\pi_1 \approx_d \pi_2$, then for all $e \in$ **Exp**, if $public(e,d)$, then $V(e,\pi_1) = V(e,\pi_2)$.*

By observing the value of declassifiable expressions, one can learn something about the actual environment. In particular one learns that it must belong to a given class of d-equivalent environments. The policy d is correctly enforced if no further information can be learned.

Definition 18 (**Revealed Knowledge** ($\mathcal{R}$))**.** *Given a declassification policy d and an environment π we define $\mathcal{R}(\pi,d) = \{\pi' | \pi \approx_d \pi'\}$.*

Note that the smaller the set $\mathcal{R}(\pi,d)$ is, the more information about π is permitted to be revealed. The revealed knowledge represents a bound on the amount of information that may be revealed by a program that complies with policy d. The next step is to define the amount of information a program actually reveals.

The behaviour of a program that an observer can see is the sequence of outputs it generates. Thus an observer cannot distinguish two environments if their runs produce the same sequence of visible output actions.

Definition 19 (**Observed Knowledge** ($\mathcal{K}$))**.** *Define $\mathcal{K}(\pi,C)$ by $\mathcal{K}(\pi,C) = \{\pi' | Run(C,\pi) \equiv_{out} Run(C,\pi')\}$.*

Our security property, Policy Controlled Release (PCR), states that the knowledge obtained from observing the program is bounded by the information released by the declassification policies.

Definition 20 (**Policy Controlled Release (PCR)**)**.** *A program C satisfies policy controlled release for policy d if for all environments $\pi : \mathcal{K}(\pi,C) \supseteq \mathcal{R}(\pi,d)$.*

VI. Soundness of the analysis

The following theorem shows that if our analysis says that a program is secure, then the program satisfies the PCR property.

Theorem 21. *For any terminating program C and a declassification policy d, if $valid(G(C),d)$ then the program C satisfies PCR.*

Proof: Lemma 24 below implies that the executions of a d-valid program in two d-equivalent environments can be linked in a way that guarantees they will result in the same runs. This implies that for all environments π,π' if $\pi' \in \mathcal{R}(\pi,d)$ then also $\pi' \in \mathcal{K}(\pi,C)$. ■

The proof of the theorem relies on a linking between runs, the existence of which is stated by Lemma 24. First we define the properties of this linking and the intuition behind how the linking works and why it must exist. The linking is inspired by the proof of soundness in Banerjee et al. [12]. However, our proof is simpler because we do not need to consider the exact path taken by the program to reach a particular state – our d-equivalence property together with our flow-sensitive approach to check validity ensures that both the runs take the same branches for paths leading to the output actions. Additionally, the proof is termination-insensitive. This means that for the proofs to go through, we assume that the loops, in which the conditional expression is non-declassifiable, terminate.

The core idea behind the linking is that a program can be in one of two distinct confidentiality levels: a level L (low, public) in which it may do output or a level H (high, secret) where it may behave differently depending on non-declassifiable information. We say that C is a *compositional statement* if C is of the form $C_1; C_2$, otherwise C is *non-compositional*. Note that any program can be written in the form $C_1; \ldots; C_n$ $(n \geq 1)$ with C_i non-compositional statements. Here we call

C_1 the active command of C, denoted $\Lambda(C)$. Given a policy d, we type all non-compositional statements contained in C as follows:

1) $\Gamma(\texttt{skip}) = H$.
2) If C_i is a conditional statement (`if` or `while`) whose condition c is not marked as declassifyable, i.e. $\neg safe((c, \texttt{var}), G(C), d)$ then $\Gamma(C_i) = H$ and also all statements nested inside C_i, directly or indirectly, are typed H.
3) If C_i is a conditional statement whose condition is declassifyable we repeat the procedure for the statement(s) in the body in the same way.
4) Each non-compositional statement not typed H according to the above rules is typed L.

The type of a compositional statement C is the type of its active command $\Lambda(C)$. We define the *low continuation of* $C = C_1; \ldots; C_n$, denoted *L-cont*(C) as the statement $C_i; \ldots; C_n$ where i is the first index for which C_i is not typed high.

In the L level the program will behave 'the same' in two d-equivalent environments. The next definitions capture this notion of 'the same'. We first consider the states that a program could reach.

Definition 22 (Compatible States ($\asymp$)). *Two states σ_1 and σ_2 are said to be compatible for program C and policy d, denoted $\sigma_1 \asymp_{(C,d)} \sigma_2$, if the following conditions hold:*

1) $\forall \alpha \in \textbf{In} : cds((\alpha, \texttt{in}), G(C), d) \Rightarrow (I_{\sigma_1}(\alpha) = I_{\sigma_2}(\alpha) \wedge PC_{\sigma_1}(\alpha) = PC_{\sigma_2}(\alpha))$.
2) $\forall x \in \textbf{Var} : cds((x, \texttt{var}), G(C), d) \Rightarrow (E_{\sigma_1}(x) = E_{\sigma_2}(x) \wedge PC_{\sigma_1}(x) = PC_{\sigma_2}(x))$.

If the control dependencies of a variable or channel are declassifiable then they cannot be altered/read from by the program in a H level and as L behaviour has to be the same, they cannot differ between two d-equivalent environments.

Definition 23 (Correspondence between two runs (Q)). *Let C be a program, π and π' be environments, and d be a policy. Let t be a prerun of $Run(C, \pi)$ and t' be a prerun of $Run(C, \pi')$ with $\|t\| = n$ and $\|t'\| = m$. A correspondence between t and t' is a relation $Q \subseteq \{1, \ldots, n\} \times \{1, \ldots, m\}$ such that $0\ Q\ 0$ and for all i, j such that $i\ Q\ j$, letting $t_i = \langle o_i, \langle C_i, \sigma_i, \pi \rangle\rangle$ and $t'_j = \langle o'_j, \langle C'_j, \sigma'_j, \pi' \rangle\rangle$, the following conditions hold:*

1) *(output-equivalence)* $\iota(t_1 \ldots t_i) = \iota(t'_1 \ldots t'_j)$
2) *(state-compatibility)* $\sigma_i \asymp_d \sigma'_j$
3) *(level-agreement)* $\Gamma(C_i) = \Gamma(C'_j)$
4) *(code-agreement L)* $\Gamma(C_i) = L \Rightarrow C_i = C'_j$
5) *(code-agreement H)* $\Gamma(C_i) = H \Rightarrow$ *L-cont*$(C_i) =$ *L-cont*(C'_j)
6) *(completeness)* $\forall i \in \{1, \ldots, n\} : \exists j \in \{1, \ldots, m\} :$ $i\ Q\ j$ *and vice versa*

We say two runs correspond if there exists a correspondence relation between them.

From the first requirement, 'output-equivalence', it is clear that two corresponding runs produce the same output. The other requirements allow us to inductively build the correspondence relation Q between runs of a program in d-equivalent environments.

Lemma 24. *Given a program C and a declassification policy d, satisfying $valid(G(C), d)$, and given two environments π and π' satisfying $\pi \approx_d \pi'$, such that the program C terminates under the environments π and π'. Let $\omega = \langle C, \sigma_{init}, \pi \rangle$, $\omega' = \langle C, \sigma_{init}, \pi' \rangle$, and S a (partial) run starting from ω then there is a partial run T starting from ω' that corresponds to ω.*

Proof: The proof, which we sketch here, inductively constructs a correspondence relation Q between the two traces. A relation Q between partial run S of length n and a corresponding run T can be extended to the partial run $S.\langle o_n, \langle C_{n+1}, \sigma_{n+1}, \pi \rangle\rangle$ of length $n + 1$ (if it exists) and an extension of run T by zero or more steps. When the active command $\Lambda(C_n)$ has level L, this is done by a straightforward case analysis of the active command in the last configuration in S. When the active command has level H, the two runs are extended until they both reach commands with level L. Extending the correspondence is straightforward in these high regions because the nondeclassifiable nature of the control context makes output impossible and maintenance of state compatibility hold trivially.

The key case in the proof handles output statements. In it, we use the induction hypothesis, together with Theorems 9 and 15, to show that output equivalence is preserved. Because of the centrality of this, we include here the proof that output equivalence is preserved in the output case. Assuming that the final element of S is $\langle o_n, \langle C_n, \sigma_n, \pi \rangle\rangle$ and the final element of T is $\langle o'_m, \langle C'_m, \sigma'_m, \pi \rangle\rangle$, in this case we have $C_n = C'_m = \gamma := x; C$ and $\Lambda(C_n) = \gamma := x$.

Since outputs happen only under declassifiable conditionals, this case can only occur when $safe((x, \texttt{var}), G(C), d)$, which means that $cds((x, \texttt{var}), G(C), d)$ and $dds((x, \texttt{var}), G(C), d)$. From the induction hypothesis, $\sigma_n \asymp_{(C,D)} \sigma'_m$. This combines with $cds((x, \texttt{var}), G(C), d)$ to give us $E_{\sigma_n}(x) = E_{\sigma'_m}(x)$. By Theorems 9, we have $E_{\sigma_n}(x) \in exp_{G(C)}((x, \texttt{var}))$. It now follows from $dds((x, \texttt{var}), G(C), d)$ by using part (i) of Theorem 15 that $public(E_{\sigma_n}(x), d)$. It now follows from $\pi \approx_d \pi'$ and Lemma 17 that $V(E_{\sigma_n}(x), \pi) = V(E_{\sigma'_m}(x), \pi')$. Therefore the outputs in both these transitions are the same. ■

VII. Related Work

Many of the initial papers on language based security [1] enforced the non-interference [2] property statically using type-based [4], [5], [20] or dataflow-analysis based [6], [7], [8], [9] approaches. Banâtre, et al. [8] were the first to propose using *accessibility graphs* to specify data and control flow dependencies between different variables in the program and thereby automatically inferring the security properties of the program. Bergeretti, et al. [21] represent information flows as relations between different variables in the program and Clark, et al. [22] represent flows as relations between the variables and the control flow points represented by the program counter.

Although the above approaches require dependency calculation similar to our expression graphs, we can additionally represent declassification policies, while they can only check for pure non-interference. More recently, Hammer, et al. [23], [24] propose an information flow control algorithm for Java. The variable dependencies are specified in the form of dependency graphs. The declassification policies are specified using path conditions, which are a conjunction of all the conditional expressions that are encountered before reaching the output program point. Although the path conditions are certainly useful to specify some kind of declassification policies, they do not compute what expressions are being declassified. Here, we attempt to capture this information using our expression graphs. Swamy, et al. [25] propose a formal language, AIR (Automata for Information Release), for describing stateful information release policies separately from the program that is to be secured. Although the policies are specified in the form of an automaton separate from the program, the approach requires that the programs be written in λAIR, a core formalism for a functional programming language, so that the AIR policies can be provably enforced.

In type-based approaches the declassification condition is tagged to the security lattice [26], [27], [28], [29] or to an expression inside the program [30]. Since declassification typically involves downgrading the security level from high to low, this is the right place to specify the policies. To specify which policy to use at the declassification points, new syntactic constructs are introduced into the programming language, making the policy and the program to be interdependent on each other. In most cases, a new *declass* command is introduced into the program. The enforcement is usually a hybrid of static analysis and dynamic execution. In some approaches [31], [32], a particular section of code is encapsulated in a conditional statement. The condition specifies the declassification policy. This section of code is executed only if the condition is true, thereby dynamically enforcing declassification. More recently, some approaches advocate specifying a special security API [13], [25], [33]. If the program is written using this API, declassification policies can be provably enforced.

Li and Zdancewic [26] use declassification policies that take the form of lambda terms over inputs, akin to our approach. Expressing the policies in lambda calculus gives them the flexibility to compare different policy terms for equivalence. This is a strength of the prior work in relation to our own. The main strength of our work in relation to theirs lies in our enforcement mechanism. For this, they use a type system that labels each variable in the program with a security policy. The security lattice is given over the lambda terms in the policy. As they also point out, their enforcement mechanism cannot handle policies such as $\lambda x : int.\lambda p : int.(x + p) * p$. On the other hand, our work handles this kind of situation, since our program expression graphs implicitly keep track of all the expressions that can flow to an output channel, enabling our approach to analyze expressions resulting from global computations. Thus, using program graphs allows us to enforce more expressive policies. The paper also hints that that their approach can be applied to untrusted code if enforced differently, but does not explain how to do so.

The type-based enforcement mechanism of delimited release [30] and localized delimited release [34] policies keep track of the variables involved the in the declassified expressions and ensure that they are not updated before declassification. This is required to prevent laundering of information. Our flow based enforcement automatically keeps track of the changes in the variables, thereby precluding the need to have an explicit declassification construct in the program.

Jif [35] is one of the most advanced programming languages designed to enforce fine-grained declassification policies in the program. However, if the programs and policy are not carefully designed, as stated in [13], there is a risk of burying the policy deep inside the code and therefore requiring a change in the program with every change in the policy. In light of this observation, several researchers studied how large programs can be written in a security typed language so that their behaviour is provably secure. Askarov, et al. [36] show how security typed languages can be used to implement cryptographic protocols and propose several design patterns to help the programmers to write their applications in Jif. They program a large poker application to demonstrate their approach. Hicks, et al. [13] propose FJifP, which includes all the security features of Jif and also an option to use certain methods as declassifiers. They also highlight the need for effective programming tools in which to write Jif programs.

Askarov, et al. [19] provided the foundation for CGR with their definition of the Gradual Release (GR) property. Their paper quantifies the knowledge obtained by the observer as the set of possible secret inputs that could be generated by observing the public outputs, i.e., the notion of *observed knowledge*. The GR property states that the observer's knowledge increases only at declassification points. Our aim of supporting policies that are as program-independent as possible prevents our considering attacker models that involve program variables other than output channels. Thus the observed knowledge in our framework is the knowledge obtained from the outputs and does not depend on any other program events. The CGR property of [12] requires the GR property. Additionally, it requires that the low-security observer of program behavior is able to detect no difference between runs that are generated from initial states that yield the same values for expressions identified in the declassification policies. Our formulations of revealed and observed knowledge follow a similar approach.

Banerjee, et al. [12] achieve separation of code and declassification policies. However, their approach does not achieve complete separation of code and policy. The *flowspecs* are a combination of a formula over program variables (P), special predicates called the agreement predicates (φ) over the program variables and a modifiable variable (x) whose type is being changed. The flowspecs are quite expressive and can be used to specify policies in *when*, *where* and *what* dimensions. However the technique only works for trusted code, which is written according to the policy specification. In their paper,

if P and φ only have global variables, then they say that x can be a schematic variable instantiated with different local variables. Although this allows them to have more flexibility in terms of applying the same policy to different parts of a large code base, it does not allow them to use the policies for entirely different programs. The policies cannot be reused for any other code in which the data structures and global variable names differ. Our policy specifications are more general and can be applied to multiple, unrelated programs.

The notion of indistinguishability used in [37] is closely related to our D-equivalence relation, as it is based on the attackers' knowledge of the initial values of high variables in their escape hatches, which resemble the declassifiable expressions identified by policy in our framework. However, their expressions are identified individually, which prevents them declassifying expressions of unbounded size, such as result from iterative computations. They also do not share our objective of completely separating policy from program. This enables them to consider where declassification occurs within the program, and to handle attacker models in which non-output events are observable, which we inherently cannot do.

Giambiagi and Dam [38] provide a framework for analyzing a security protocol's implementation against its specification. A dependency specification defines an information flow property by characterizing the direct flow along a path in the form of allowed sequence of API and primitive function calls. However, as the authors mention in the paper, dependency specifications are very low-level objects, which can be used as intermediate representations of flow requirements. In general, their dependency specifications should accurately capture the exact number of times a method is called during a particular flow and it can only characterize a single flow. By contrast, our expression graph representation can represent several flow patterns, including loops.

Taint analysis [39], [40] considers direct data flows, but, unlike information flow analysis, ignores control flows. In this sense, it is much less demanding than declassification-policy enforcement.

Giacobazzi and Mastroeni [41] provide a powerful framework in which to specify the weakened variant of non-interference that is enforced under a declassification policy. We think its likely that our Policy Controlled Release property could be precisely stated in their framework, modulo the fact that our approach is communication channel-oriented, while theirs focuses on state transformation. We view our contribution as bringing the field closer to being able to implement a large class of practical analyses that can be specified in their framework. This prior work is highly abstract, and provides little guidance with respect to the construction of usable analysis tools.

VIII. Discussion

Our approach to policy specification and enforcement achieves a clean separation of code and policy. Because of this separation, the system operator has the ability to select different policies for different situations/uses, and to determine through the use of our analysis whether a given program is appropriate for each use. An additional advantage of our framework is the ability to handle declassification policies depending on recursive structures such as loops.

We have identified two issues that remain to be addressed. First, as illustrated in the discussion at the end of Section II, one would like to be able to express requirements regarding the number of times a loop runs before declassifying the resulting value, for instance, to ensure that enough values are being averaged. We believe that a full treatment of the problem of how many times a loop runs can be achieved via a cooperation between static analysis and runtime enforcement mechanisms, with the former specifying the constraint and the latter enforcing it. Secondly, this paper does not address the problem of matching program graphs against policy graphs up to algebraic equivalence of the denoted expressions [42]. This would enable verification of programs that compute expressions that differ from those identified by the declassification policy, but that are equivalent to them under algebraic laws, such as associativity, commutativity, and idempotence. One possible approach to this would be to transform program and policy graphs into a normal form and then apply the matching we propose here. Of course, in general, this can lead to a combinatorial explosion, and some conservative approximation may be necessary to retain tractability.

Our flow-analysis is termination-insensitive [43]. We can make the analysis termination-sensitive either by disallowing while loops under high conditionals or introducing a flow between the conditional in which the loop is declared to all the output channels in the program. However, both these approaches are too restrictive. Disallowing the while loops under high conditionals would also make the program dependent on the policy, which we want to avoid. Achieving the right balance between handling termination behavior correctly versus ensuring that the analysis is practical is tough and it is out of the scope of the current paper. In this paper, therefore, we do not deal with termination and timing channels.

The policies used by our framework specify the expressions over inputs that can be declassified, so they address the *what* dimension of declassification. It is straightforward to extend our analysis to address the *who* dimension, as the system operator can control which policy graphs are used in analyzing the program based on *who* wrote the program and the policies, and *who* is going to observe the outputs from the output channels. On the other hand, utilizing the *where* dimension extensively would be contrary to our goal of making the policy program-independent. In the case of legacy code, the programs are typically written without information-flow policies explicitly defined. For untrusted code, we have sought an approach that provides assurance without requiring to trust the programmer. Nevertheless, for cases in which the *where* dimension is required, it is straightforward to specify program points at which a particular policy may be applied by associating this condition with the policy itself; no code-annotations are required. The *when* dimension may entail that some part of the program be verified with one policy, and other

parts be verified with another, based on a condition that might occur during the execution of the program. Specifying such policies require intimate knowledge of the program and might be possible for trusted code which is being newly written. For legacy and untrusted code, it is unrealistic to specify such a dimension.

One premise of our work is that declassifications are not invertible. This assumption is realistic since we consider well-formed policies. If a declassification policy allows a $f(\alpha)$, in the scenario where a f^{-1} function exists, then the policy is actually allowing α to be disclosed. Even if we had some mechanism that checks if a given function had not been inverted throughout the code, nothing would prevent the inverse function from being applied outside of the program. Therefore, we assume that well-formed policies do not allow invertible expressions to be declassified.

Sections V and VI assume that all output channels in the program are observable. We also assume that all the input channels are controlled by the target system. To weaken these requirements, we can associate security levels to inputs and outputs, and specify allowed flows by using standard lattice models. The labels on policy graph nodes would be extended accordingly, and the analysis algorithm would be required to respect these labels. This would result in very little change to our approach or in the PCR theorem and its proof.

If the input channels used in the program are interactive, the values of the inputs can change outside the program control. As a result, accessing one input channel may influence the value of another input channel. Such interactions are not considered in our current threat model. However, they can be treated by including additional assumptions about the interactions between the channels. For example, if a collection of input channels is under an attacker's control, they are somewhat equivalent to each other, thus requiring that all such channels share the same collection of control dependencies. Thus, if any of them is read in a non-declassifiable context, no value read from any of these channels may be declassified. A similar reasoning applies if reads themselves are observable events. We plan to address these issues in future work.

Our approach can also be extended to support a broader range of language constructs. Since we rely on ϕ-functions from SSA translation to recognize control-flow branches, commands like `case`, `continue`, `break` and others can be included in our mechanism in a straightforward manner, provided there is a valid translation of them to SSA form. For constructs such as procedures, methods, classes and inheritance, our approach can be adapted to work in a modular way [44]. Individual blocks of code, such as user defined functions, can be analyzed by generating separate graphs, and calls to these blocks would use "procedure call edges" to reference "argument nodes". Global variables (and class parameters), however, would need special treatment. As with most language-based information flow techniques, extending our approach to allow concurrency and constructs that enable control flow to jump to unpredictable points of the code (e.g., computed `goto`, exceptions) pose bigger challenges.

ACKNOWLEDGEMENT

Part of this research was supported by the STW Sentinels Project "S-Mobile", the NSF under awards CNS 08-31212, CNS-0716210 and CNS-0716750 and the Texas Higher Education Coordinating Board NHARP Award 010115-0037-2007. We would like to thank all the reviewers for their helpful comments and suggestions.

REFERENCES

[1] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, 2003.

[2] J. A. Goguen and J. Meseguer, "Security policies and security models," in *SP '82: Proceedings of the 3rd IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[3] A. C. Myers, "JFlow: practical mostly-static information flow control," in *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1999, pp. 228–241.

[4] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 1, pp. 117–158, 2003.

[5] D. M. Volpano, C. E. Irvine, and G. Smith, "A Sound Type System for Secure Flow Analysis," *Journal of Computer Security*, vol. 4, pp. 167–188, 1996.

[6] T. Amtoft and A. Banerjee, "Information Flow Analysis in Logical Form," in *SAS '04: 11th International Static Analysis Symposium*, 2004, pp. 100–115.

[7] T. Amtoft, S. Bandhakavi, and A. Banerjee, "A logic for information flow in object-oriented programs," in *POPL '06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, pp. 91–102.

[8] J.-P. Banâtre, C. Bryce, and D. L. Métayer, "Compile-Time Detection of Information Flow in Sequential Programs," in *ESORICS '94: Proceedings of the 3rd European Symposium on Research in Computer Security*. London, UK: Springer-Verlag, 1994, pp. 55–73.

[9] D. E. Denning, "A Lattice Model of Secure Information Flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[10] A. Sabelfeld and D. Sands, "Dimensions and Principles of Declassification," in *CSFW '05: Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, 2005, pp. 255–269.

[11] S. Zdancewic, "Challenges in Information-flow Security," in *PLID '04: Proceedings of the First International Workshop on Programming Language Interference and Dependence*, Verona, Italy, August 2004.

[12] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive Declassification Policies and Modular Static Enforcement," in *SP '08: Proceedings of the 29th IEEE Symposium on Security and Privacy*, 2008, pp. 339–353.

[13] B. Hicks, D. King, P. McDaniel, and M. Hicks, "Trusted declassification: high-level policy for a security-typed language," in *PLAS '06: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*. New York, NY, USA: ACM, 2006, pp. 65–74.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[15] G. Bilardi and K. Pingali, "Algorithms for computing the static single assignment form," *Journal of the ACM*, vol. 50, no. 3, pp. 375–425, 2003.

[16] M. M. Brandis and H. Mössenböck, "Single-pass generation of static single-assignment form for structured languages," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1684–1698, 1994.

[17] R. Milner, *Communication and concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[18] B. P. S. Rocha, S. Bandhakavi, J. den Hartog, W. H. Winsborough, and S. Etalle, "Towards Static Flow-based Declassification for Legacy and Untrusted Programs," Tech. Rep., to appear.

[19] A. Askarov and A. Sabelfeld, "Gradual Release: Unifying Declassification, Encryption and Key Release Policies," in *SP '07: Proceedings of the 28th IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 207–221.

[20] A. Banerjee and D. A. Naumann, "Secure Information Flow and Pointer Confinement in a Java-like Language," in *CSFW '02: Proceedings of the 15th IEEE Workshop on Computer Security Foundations*. Washington, DC, USA: IEEE Computer Society, 2002, p. 253.

[21] J.-F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 37–61, 1985.

[22] D. Clark, C. Hankin, and S. Hunt, "Information flow for algol-like languages," *Computer Languages*, vol. 28, no. 1, pp. 3–28, 2002.

[23] C. Hammer, J. Krinke, and G. Snelting, "Information flow control for Java based on path conditions in dependence graphs," in *ISSSE '06: Proceedings of the IEEE International Symposium on Secure Software Engineering*. IEEE, 2006.

[24] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, vol. 8, no. 6, pp. 399–422, December 2009, supersedes ISSSE and ISoLA 2006.

[25] N. Swamy and M. Hicks, "Verified enforcement of stateful information release policies," in *PLAS '08: Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. New York, NY, USA: ACM, 2008, pp. 21–32.

[26] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," *SIGPLAN Notices*, vol. 40, no. 1, pp. 158–170, 2005.

[27] S. Chong and A. C. Myers, "Security policies for downgrading," in *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2004, pp. 198–209.

[28] S. Tse and S. Zdancewic, "A Design for a Security-Typed Language with Certificate-Based Declassification," in *ESOP '05: 14th European Symposium on Programming*, 2005, pp. 279–294.

[29] S. Chong and A. C. Myers, "End-to-End Enforcement of Erasure and Declassification," in *CSF '08: Proceedings of the 21st IEEE Computer Security Foundations Symposium*, 2008, pp. 98–111.

[30] A. Sabelfeld and A. C. Myers, "A Model for Delimited Information Release," in *ISSS '03: International Symposium on Software Security*, 2003, pp. 174–191.

[31] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic, "Managing policy updates in security-typed languages," in *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, 2006, pp. 202–216.

[32] S. Bandhakavi, W. H. Winsborough, and M. Winslett, "A Trust Management Approach for Flexible Policy Management in Security-Typed Languages," in *CSF '08: Proceedings of the 21st IEEE Computer Security Foundations Symposium*, 2008, pp. 33–47.

[33] B. Hicks, D. King, and P. McDaniel, "Declassification with Cryptographic Functions in a Security-Typed Language," Network and Security Center, Department of Computer Science, Pennsylvania State University, Tech. Rep. NAS-TR-0004-2005, January 2005, (*updated May 2005*).

[34] A. Askarov and A. Sabelfeld, "Localized delimited release: combining the what and where dimensions of information release," in *PLAS '07: Proceedings of the 2nd Workshop on Programming Languages and Analysis for Security*. New York, NY, USA: ACM, 2007, pp. 53–60.

[35] S. Chong, A. C. Myers, K. Vikram, and L. Zheng, *Jif Reference Manual*, June 2006. [Online]. Available: http://www.truststc.org/pubs/548.html

[36] A. Askarov and A. Sabelfeld, "Security-Typed Languages for Implementation of Cryptographic Protocols: A Case Study," in *ESORICS '05: Proceedings of the 10th European Symposium on Research in Computer Security*, 2005, pp. 197–221.

[37] ——, "Tight enforcement of information-release policies for dynamic languages," in *CSF '09: Proceedings of the 22nd IEEE Computer Security Foundations Symposium*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 43–59.

[38] P. Giambiagi and M. Dam, "On the secure implementation of security protocols," *Science of Computer Programming*, vol. 50, no. 1-3, pp. 73–99, 2004.

[39] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *SSYM '05: Proceedings of the 14th Conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2005, pp. 18–18.

[40] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: effective taint analysis of web applications," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2009, pp. 87–97.

[41] R. Giacobazzi and I. Mastroeni, "Abstract non-interference: parameterizing non-interference by abstract interpretation," in *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2004, pp. 186–197.

[42] J. Hendrix and H. Ohsaki, "Combining equational tree automata over ac and aci theories," in *RTA '08: Proceedings of the 19th International Conference on Rewriting Techniques and Applications*. Springer-Verlag, 2008, pp. 142–156.

[43] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*. Springer, 2008, pp. 333 – 348.

[44] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1995, pp. 49–61.