Enforcing Trace Properties by Program Transformation

Thomas Colcombet Pascal Fradet

Ens Lyon Inria

Irisa
Campus de Beaulieu, 35042 Rennes Cedex, France
{colcombe,fradet}@irisa.fr

Abstract

We propose an automatic method to enforce trace properties on programs. The programmer specifies the property separately from the program; a program transformer takes the program and the property and automatically produces another "equivalent" program satisfying the property. This separation of concerns makes the program easier to develop and maintain. Our approach is both static and dynamic. It integrates static analyses in order to avoid useless transformations. On the other hand, it never rejects programs but adds dynamic checks when necessary. An important challenge is to make this dynamic enforcement as inexpensive as possible. The most obvious application domain is the enforcement of security policies. In particular, a potential use of the method is the securization of mobile code upon receipt.

1 Motivation and approach

Programming can be seen as the task of implementing a collection of properties (the specification). Let us focus on two problems the programmer may face in this task.

- *The expressivity problem.* Available programming languages make some properties very difficult to implement. There are no suitable abstractions to represent them and their implementation is scattered throughout the program. Security or robustness properties are two striking examples. Enforcing such properties may involve inserting checks all over the program.
- *The correctness problem.* Two kinds of approaches have been considered to prove/ensure that a program satisfies a property. Static approaches such as formal program derivation, disciplines of programming (e.g. enforced by a type system or a domain-specific language), and static program analyses can ensure properties without any runtime penalty. However, they are either very costly, dedicated to a specific property, or reject perfectly correct programs without returning any useful feedback. Dynamic (or system) approaches, such as monitors or security kernels, enforce the property based on some kind of state machine that evolves and checks the property in parallel with program execution. The main drawback of this approach is its runtime cost.

To appear in the proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 2000.

In this paper, we advocate another approach, reminiscent of aspect-oriented programming [11], that overcomes the expressivity and correctness problems for a particular class of properties.

The property is expressed separately from the program at a specification level. A program transformer takes the program and the property, and automatically produces another "equivalent" program respecting the property. The transformed program is equivalent to the original program except for inputs for which the source program violates the property. In this case, the transformed program produces an exception and stops. An important challenge is to make this dynamic enforcement as inexpensive as possible. In particular, if we are able to detect statically that the source program satisfies the property, then no transformation should be done.

A particularity of our proposal is that it combines a static and a dynamic approach. It integrates static analyses in order to avoid unnecessary runtime cost. On the other hand, it does not reject programs but adds dynamic checks when necessary. Another significant feature is its flexibility: programmers can specify their own properties in a declarative way.

We consider here safety properties expressed on the execution traces of programs. The most obvious application domain is the enforcement of security policies. Many security policies can be specified as properties on traces. Let us cite, for example, access-control models such as the high-water-mark model, or the Chinese wall policy [1]. Clearly, other kinds of properties, such as some safety or robustness properties, can be expressed as trace properties.

A potential use of the method is the securization of mobile code upon receipt. Standard byte code contains enough structure and information to make the application of our method realistic. The benefit of such a just-in-time transformation is that it could be done according to customized security properties. This could be seen as a flexible and simple alternative to proof-carrying code [16] for the class of properties considered here.

2 Overview

Roughly speaking, we start with a program P, and a trace property T and produce a transformed program $Trans[P, T]$

$$P \equiv \begin{cases} \texttt{manager();} \\ \texttt{if(...) accountant();} \\ \texttt{if(...) \{critical();} \\ \qquad \texttt{manager() ;\}} \\ \texttt{accountant();} \\ \texttt{critical();} \end{cases} \qquad E \equiv \begin{cases} \texttt{manager(*)} \;\texttt{->}\; \texttt{m} \\ \texttt{accountant(*)} \;\texttt{->}\; \texttt{a} \\ \texttt{critical(*)} \;\texttt{->}\; \texttt{c} \end{cases} \qquad T = ((a^+m \mid m^+a)(a \mid m)^*c)^*$$

$$\mathit{Trans}[P,T] \equiv \begin{cases} \texttt{state = 0;} \\ \texttt{manager();} \\ \texttt{if(...) \{state=1;} \\ \qquad \texttt{accountant();\}} \\ \texttt{if(...) \{if (state == 0) \{abort;\}} \\ \qquad \texttt{critical();} \\ \qquad \texttt{manager();\}} \\ \texttt{accountant();} \\ \texttt{critical();} \end{cases}$$

Figure 1: A small example of property enforcement

respecting T. If the execution trace of (P, σ_0) (the source program with some initial store) respects the property then $(\mathit{Trans}[P,T], \sigma_0)$ yields the same results. Otherwise, $(\mathit{Trans}[P,T], \sigma_0)$ produces an exception and stops just before violating the property. In fact, we also take as input a function E mapping instructions of P to events of interest. This function is the formal link between the program and the property that is expressed on these events. Figure 1 presents a simple example where the events of interest `m, a, c` are calls to the procedures `manager`, `accountant`, and `critical`. The property T, expressed as a regular expression, requires that events `m` and `a` take place before each event `c`. That is to say, a critical action cannot take place before the clearance of the manager and the accountant (segregation of duties). The source program may violate this property whenever the first conditional is false. The transformed program, where two assignments and a conditional have been inserted, respects the property (i.e. aborts whenever the property is about to be violated).
Our approach comprises seven phases.

1. *Property encoding.* We consider only safety properties; (i.e. properties stating that no "bad thing" happens). Liveness properties ("good things" do happen) cannot be dynamically enforced ([18],[19]). A safety (or enforceable) property can be characterized by a set of disallowed finite executions. We restrict ourselves to regular safety properties (i.e. the set of disallowed executions is regular). The language to express such properties can be based on logic (e.g. LTL [4] or WS1S [13]) or regular expressions. The important point for us is that the property can be encoded by a finite state automaton. The language recognized by the automaton will be the set of all authorized partial traces of events.

2. *Program annotation.* The first phase is to locate and annotate the events of interest in the program. These events can be assignments to specific variables, calls to specific methods, opening of files, etc. All irrelevant instructions are associated with the dummy event $\star$. A key constraint is that a program instruction is associated with only one event. This is easy to ensure when events are specified based solely on the program syntax (e.g. the function E in Figure 1). But, one would also like to express properties on "semantic" events such as "*x is assigned the value 0* ". In general, one cannot decide statically whether an assignment `x := e;` will generate this event or not. In order to take semantic events into account, the program must be transformed beforehand. For that example, each assignment `x:=e` could be transformed into `if e=0 then x:=e else x:=e`[1] so that instructions are associated with a single event. It is always possible to insert such tests automatically; the real challenge for this phase is to avoid inserting spurious tests. For that reason, such pre-transformations must rely on static program analyses.

3. *Program abstraction.* The program is abstracted into a graph whose nodes denote program points and edges represent instructions. This phase makes the core of the approach independent of the programming language. In this paper, we concentrate on control-flow graphs and trace properties. Other abstractions can be accommodated without changing the core of the approach. In order to produce a precise abstraction, this phase may rely on a static analysis (e.g. a control-flow analysis). The important point is that the set of actual traces (restricted to events) is a subset of the traces generated by the graph (i.e. the abstraction is safe). For the sake of clarity, the technical part of the paper is presented with simple (intraprocedural) graphs. The extension to interprocedural (or context-free) graphs is outlined in Section 4. We mention in Section 5.2 another abstraction (call graphs) where new properties (stack properties), which would not be regular over standard traces, can be expressed.

4. *Direct instrumentation.* The next phase is to transform the graph in order to remove from its set of partial traces those forbidden by the automaton. A natural idea would be to consider the program graph as an automaton and express the transformation as a product of automata followed by optimizations. This idea is not satisfactory for two reasons. Firstly, since the objective is to map the transformed graph back to a program, the

[1] Note that this code will be transformed by subsequent phases before the compiler gets a chance to transform it back to `x:=e`.

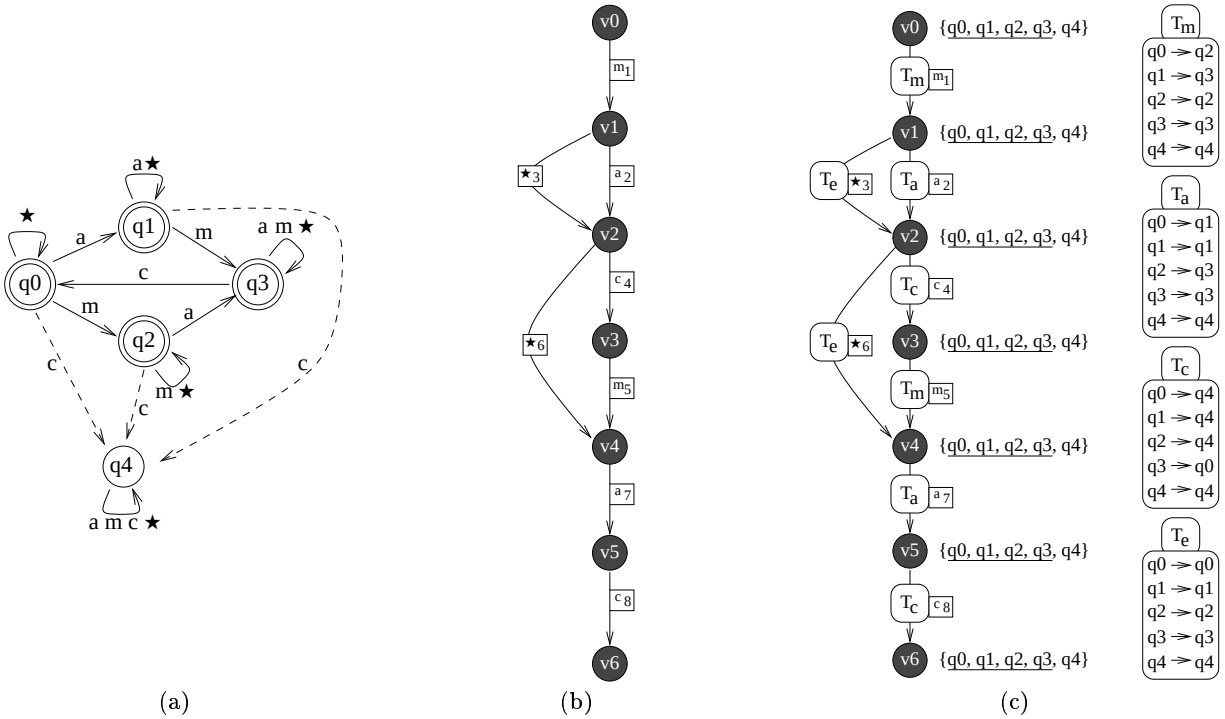


Figure 2: Automaton (a), control-flow graph (b) and direct instrumentation (c)

representation must remain as close as possible to the source graph. Secondly, while intraprocedural graphs are similar to automata, this is not the case when interprocedural control-flow is taken into account.

We choose to represent the integration of an automaton into a program graph by an instrumented graph or $\mathcal{I}$-graph. A $\mathcal{I}$-graph is a graph equipped with additional structures (states and transition functions) to mimic the automaton execution. We begin by a direct instrumentation that is optimized by the two subsequent phases.

5. *Minimization.* The direct instrumentation is first specialized to the program. This phase involves computing the set of reachable states for each node (this may avoid inserting useless checks in the program later on) and a transformation similar to the standard automaton minimization.

6. *Erasing.* The erasing phase transforms the minimized $\mathcal{I}$-graph in order to turn as many transitions as possible into the identity function. This phase involves several static analyses: choosing the transitions to erase, finding a new numbering of states, and computing the new $\mathcal{I}$-graph. Erasing reduces the number of state evolutions during the execution (i.e. the number of assignments that would be inserted in an imperative program).

7. *Concretization.* The optimized $\mathcal{I}$-graph must be turned into a program. It is still very close to the source program: its nodes and edges represents program points and instructions. Like the abstraction, the concretization is language dependent. In any case, one needs a way to store, fetch, and test a value (the automaton state) without affecting the source program. One also need a way to abort the program. In languages with side-effects, this can be done by local transformations (e.g. inserting assignments, conditionals). In a pure functional language, it would involve a global transformation in order to thread the automaton state throughout the execution.

In this extended abstract, we focus on the core of the method, that is to say, Phases 4, 5, and 6 (Section 3). Section 4 is devoted to the extension of the abstract framework to the interprocedural case. In Section 5, we present, independently of a particular programming language, the important properties that the abstraction (Phase 3) and the concretization (Phase 7) must satisfy in order to ensure the correctness of the global transformation. Phases 1 and 2 are not described any further.

Many of our definitions, functions, or transformations are relative to structures (automaton, graph, $\mathcal{I}$-graph) left implicit in notations. The transformations deal with a single structure and no confusion should arise.

3 Abstract framework

The abstract framework allows us to define our approach in a generic way. We start by presenting the abstract representations of programs, properties, and transformed programs. We proceed by the description of the three steps leading to the optimized instrumented graph.

3.1 Abstract representations

Trace properties as automata. We represent a trace property by a finite state automaton $\mathcal{A} = (Q, q_0, A, \Sigma, \delta)$ where Q is a finite set of states, q_0 the initial state, A the set of final (accepting) states, Σ the alphabet made of events, and δ the transition function. Any string which does not violate the property is accepted by the automaton. In practice, this implies that all states will be final except a single, trap state. As soon as this trap state is reached, the property is violated and the automaton will remain in this state.

The set of event strings verifying the property is defined by

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta(q_0, w) \in A\}$$

where, as it is common, the transition function δ has been extended from letters to strings.

Figure 2a presents the automaton derived from property T of Figure 1. A transition with the dummy event $\star$ has been added for each state. Whenever the property is violated, the automaton goes and remains, in the trap state q_4.

Programs as graphs. We represent a program as a rooted graph $\mathcal{G} = (V, v_0, E)$. Intuitively, vertices (V) represent program points, the root ($v_0 \in V$) denotes the first program point, whereas edges ($E \subseteq V \times \Sigma \times \mathbb{N} \times V$) represent instructions generating events of interest ($\in \Sigma$). We say that an edge (v, a, k, v') or $v \xrightarrow{a_k} v'$ generates the annotated event a_k; the integer k serves to identify the instruction producing the event. We extend the notation to paths and write $v \xrightarrow{w} v'$ for a path from v to v' generating the string of annotated events w ($\in (\Sigma \times \mathbb{N})^*$). The set of all partial traces generated by a graph $\mathcal{G}$ is defined as

$$\mathcal{T}(\mathcal{G}) = \{w \in (\Sigma \times \mathbb{N})^* \mid \exists v \in V,\ v_0 \xrightarrow{w} v\}$$

We write $\overline{w}$ for the string of events obtained by getting rid of the integers in w.

Figure 2b presents the rooted graph derived from the program P of Figure 1. Control-flow edges corresponding to irrelevant instructions bear the dummy event $\star$.

Transformed programs as $\mathcal{I}$-graphs. Given a graph $\mathcal{G}$ (i.e. an abstract program) and an automaton $\mathcal{A}$ (i.e. a trace property) sharing the same event set Σ, we seek to derive a new graph which generates the same set of partial traces as $\mathcal{G}$ except those invalidated by the automaton. The integration of an automaton in a graph is represented by an *instrumented graph* (or *$\mathcal{I}$-graph*).

A $\mathcal{I}$-graph $\mathcal{I} = (\mathcal{G}, R, r_0, \gamma, S)$ is made of a graph $\mathcal{G}$, a set of states R, an initial state $r_0 \in R$, a collection of final states sets $S = \{A_v \subseteq R \mid v \in V\}$, and a transition function $\gamma : (\Sigma \times \mathbb{N}) \to R \to R$. We write γ_e (instead of $\gamma(e)$) for the transition function associated with the annotated event e.

The instrumentation encodes the states and the transitions of the automaton. Each graph node is equipped with a set of final states (accepting states) and each annotated event comes with a transition function which will make the automaton state evolve according to the event.

The set of partial traces of the $\mathcal{I}$-graph $\mathcal{I}$ is defined as

$$\mathcal{T}(\mathcal{I}) = \{w \in (\Sigma \times \mathbb{N})^* \mid \exists v \in V,\ v_0 \xrightarrow{w} v \wedge \gamma_w(r_0) \in A_v\}$$

where the transition function γ has been extended to strings ($\gamma_{wv} = \gamma_v \circ \gamma_w$).

DEFINITION **1** *An instrumented graph $\mathcal{I} = (\mathcal{G}, R, r_0, \gamma, S)$ is said to enforce the automaton $\mathcal{A}$ iff*

$$\mathcal{T}(\mathcal{I}) = \{w \in \mathcal{T}(\mathcal{G}) \mid \overline{w} \in \mathcal{L}(\mathcal{A})\}$$

3.2 The direct instrumentation

The direct instrumentation of a graph $\mathcal{G}$ and an automaton $\mathcal{A}$ takes the set of states of $\mathcal{A}$ as its set of states, the initial state of $\mathcal{A}$ as its initial state, the set of final states of $\mathcal{A}$ as the set of accepting states for each node, and associates each edge with the transition function of $\mathcal{A}$ specialized to the edge event. Formally, $\mathcal{I}^d = (\mathcal{G}, Q, q_0, \gamma, S)$ with:

- $S = \{A_v = A \mid v \in V\}$
- $\gamma = \lambda e.\lambda s.\delta(s, \overline{e})$

It is easy to check that the direct instrumentation enforces $\mathcal{A}$ (Definition 1).

Figure 2c presents the direct instrumentation of the previous graph and automaton. Edges are decorated by a transition table and nodes by a set of states where accepting (final) states are underlined.

This solution can be seen as a transformation-based implementation of system kernel approach. Each event of interest involves a state transition of the underlying automaton. For an imperative program, this would mean an assignment and a test before each program instruction generating an event. It is obviously much more costly than necessary. We focus now on optimizations of the direct instrumentation.

3.3 Minimization

The automaton specifies a general property independently of programs. A program may obey the property and no transformation is needed in this case. So, the first step is to specialize the automaton to the program. We compute, using a standard fix-point iteration, the set of reachable states at each program point:

$$\forall v \in V \quad reach(v) = \{\gamma_w(r_0) \mid v_0 \xrightarrow{w} v\}$$

Detecting that the trap state cannot be reached at a program point permits to save a test.

We can now minimize the underlying automaton. This idea is similar to the standard automaton minimization: if the language recognized starting from two states is the same then the states can be merged. In our case, such a language is the set of authorized partial traces starting from a node v in a state r. This language is written $\mathcal{T}(\mathcal{I}[v, r \mapsto v_0, r_0])$ where $\mathcal{I}[v, r \mapsto v_0, r_0]$ is the instrumented graph $\mathcal{I}$ where the root has been changed to v and the initial state to r. Two states reachable at node v are considered equivalent (and can be merged) if the language generated starting from v is the same for both states. Formally,

$$r_1 \overset{\mathcal{I}}{\equiv}_v r_2 \Leftrightarrow \mathcal{T}(\mathcal{I}[v, r_1 \mapsto v_0, r_0]) = \mathcal{T}(\mathcal{I}[v, r_2 \mapsto v_0, r_0])$$

The relation $\overset{\mathcal{I}}{\equiv}_v$ is an equivalence relation and we write $class^v_{\mathcal{I}}(r)$ for the equivalence class of r. Given an instrumented graph $\mathcal{I}$, its minimization is written $\mathcal{I}^m$ with $\mathcal{I}^m = (\mathcal{G}, R^m, r_0^m, \gamma^m, S^m)$ and:

- $R^m = \bigcup_{v \in V} \{class^v_{\mathcal{I}}(r) \mid r \in reach(v)\}$

- $r_0^m = class_{\mathcal{I}}^{v_0}(r_0)$
- $S^m = \{A_v^m \mid v \in V\}$
  $with\ A_v^m = \{class_{\mathcal{I}}^v(r) \mid r \in A_v \cap reach(v)\}$
- $\forall v \xrightarrow{e} v' \in E,\ \forall r \in reach(v),$
  $\gamma_e^m(class_{\mathcal{I}}^v(r)) = class_{\mathcal{I}}^{v'}(\gamma_e(r))$

The states are now the equivalence classes of reachable states, the new initial state is the equivalence class of the initial state, the new final states are the equivalence classes of the reachable final states, and, for each edge $v \xrightarrow{e} v'$, the transition function maps the equivalent class of each reachable state r in v to the equivalent class of $\gamma_e(r)$ in v'.

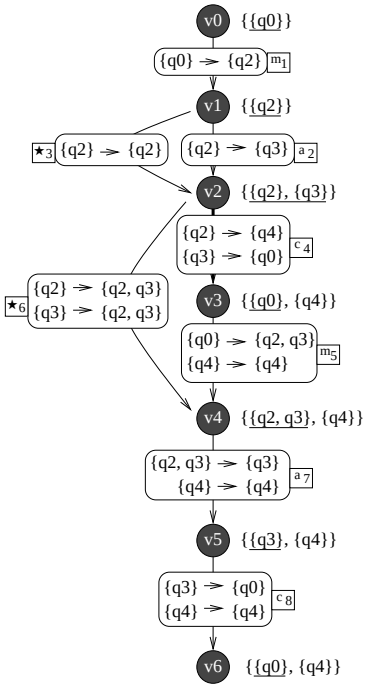


Figure 3: Minimization

The minimization of the $\mathcal{I}$-graph of Figure 2c is presented in Figure 3. The states q_2 and q_3 have been merged at node v_4 (all traces starting from v_4 in the state q_2 or q_3 are authorized). They are represented by the single state written $\{q_2, q_3\}$.

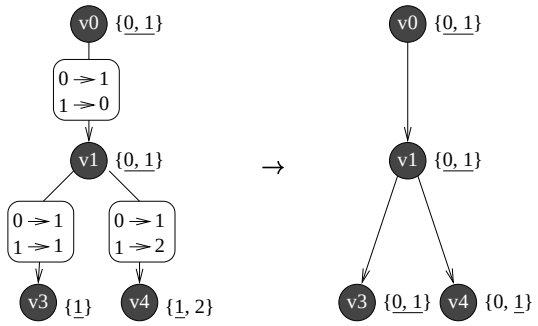
The minimization process does not change the semantics of a $\mathcal{I}$-graph.

PROPERTY **2** *For all instrumented graph* $\mathcal{I}$, $\mathcal{T}(\mathcal{I}) = \mathcal{T}(\mathcal{I}^m)$

As for automata minimization, the minimal $\mathcal{I}$-graph is unique up to isomorphism. After minimization, each node of the instrumentation of a graph obeying the property will have a single reachable state. But the main benefit of minimization is to make the next transformation more effective and to lead to optimal instrumentations (see Property 5).

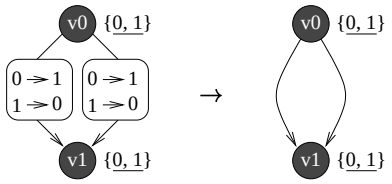
3.4 Transition erasing

Transitions associated with edges represent the evolution of states and will entail new code to be introduced in the program (e.g. assignments in an imperative program). We now strive to associate as many edges as possible with the identity function. We call this transformation *transition erasing*. Let us take a few basic examples to illustrate the idea of erasing.



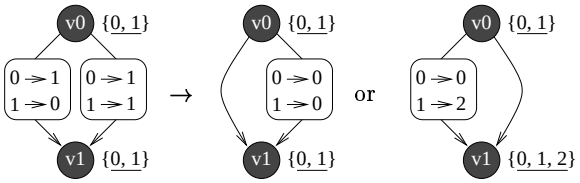
All the transitions on the original (left) $\mathcal{I}$-graph can be replaced by identity transitions. In order to keep the same language of traces, accepting states must be changed (see, for example, the node v_4). Also, some states may have to be added at a node (e.g. both states 0 and 1 at node v_3 in the right $\mathcal{I}$-graph represents 1 in the left one). This erasing of transitions can be done for any tree shaped graph.

The same idea can also be applied to other types of graphs, as the following example shows.



The two transitions are equal and it is not necessary to distinguish the two paths. Both transitions can be erased simultaneously.

Of course, not every transition can be erased.



In this case, we may choose to erase the left or the right transition, but not both.

In general, we cannot erase transitions when two paths from the same node v to the same node v' map the same state r to two different states r_1 and r_2[2]. If one associates every edge of these paths with the identity transition, there is no way to differentiate r_1 and r_2 at v' anymore. This would be incorrect since the languages $\mathcal{T}(\mathcal{I}[v', r_1 \mapsto v_0, r_0])$ and $\mathcal{T}(\mathcal{I}[v', r_2 \mapsto v_0, r_0])$ might be different. Actually, after minimization, these languages *must* be different. This is

[2] Typically, this case occurs for the control flow of conditionals and loops.

the reason why minimization improves the suppression of transitions. Without it, we would not suppress transition functions in the above case even when it is perfectly correct to do so (i.e. when $\mathcal{T}(\mathcal{I}[v', r_1 \mapsto v_0, r_0]) = \mathcal{T}(\mathcal{I}[v', r_2 \mapsto v_0, r_0])$).

The notion of *free set* of edges formalizes the previous remarks.

DEFINITION **3** *A set $F \subseteq E$ is said to be free for $\mathcal{I} = (\mathcal{G}, R, r_0, \gamma, S)$ if:*

$$\forall\ v \xrightarrow{w_1} v',\ v \xrightarrow{w_2} v'\ \in F^*,\ \forall r \in reach(v),\ \gamma_{w_1}(r) = \gamma_{w_2}(r)$$

Every edge belonging to a free set can be associated with the identity transition function. We postpone the discussion about the computation of a free set to the end of this section. For now, we assume that we have a free set F and we describe how to transform $\mathcal{I}$ into an equivalent instrumentation $\mathcal{I}^f$ when each edge in F is associated with the identity transition.

The transformation relies on a function Γ mapping the states of $\mathcal{I}$ to set of states of $\mathcal{I}^f$. A state r at a node v of $\mathcal{I}$ will be represented by the set of nodes $\Gamma(v, r)$ in $\mathcal{I}^f$. For example, if two paths of F^* reach the node v with state r in $\mathcal{I}$ then both paths will reach the node v with states (not necessarily the same) belonging to $\Gamma(v, r)$.

The function Γ is defined using constraints. First, since we want to turn transitions into identity, the set representing a state r in a node v must be included in the sets representing the image of r by these transitions.

$$\forall\ v \xrightarrow{e} v' \in F,\ \forall r \in reach(v),\ \Gamma(v, r) \subseteq \Gamma(v', \gamma_e(r)) \quad (1)$$

Second, the function Γ should not introduce state confusion. If r_1 and r_2 are different states at a node v in $\mathcal{I}$ then their representations in $\mathcal{I}^f$ must be disjoint.

$$\forall v \in V,\ \forall r_1, r_2 \in reach(v),$$
$$r_1 \neq r_2\ \Rightarrow\ \Gamma(v, r_1) \cap \Gamma(v, r_2) = \emptyset \quad (2)$$

This is not yet sufficient since the function $\Gamma(v, r) = \emptyset$ for all v and r would be a solution. Each state of a node v in $\mathcal{I}$ must have at least one image in the sates of v in $\mathcal{I}^f$. This is achieved by choosing a default value $c_{v,r}$ for each $\Gamma(v, r)$.

$$\forall v \in V,\ \forall r \in reach(v), \exists\ c_{v,r} \in \Gamma(v, r) \quad (3)$$

The smallest function Γ respecting the constraint (1) can be computed by a fix-point iteration. If we take a set of initial values $c_{v,r}$ such that $(v, r) \neq (v', r') \Rightarrow c_{v,r} \neq c_{v',r'}$ then no state confusion can occur and constraint (2) will be verified.

Given a function Γ, a $\mathcal{I}$-graph $\mathcal{I}$ is transformed into $\mathcal{I}^f = (\mathcal{G}, R^f, r_0^f, \gamma^f, S^f)$ with:

- $R^f = \{c_{v,r}\ \mid\ v \in V,\ r \in reach(v)\}$
- $r_0^f = c_{v_0, r_0}$
- $\begin{cases} \forall\ v \xrightarrow{e} v'\ \in F,\ \gamma_e^f = Id_{R^f} \\ \forall\ v \xrightarrow{e} v'\ \notin F, \forall\ r \in reach(v),\ \forall r^f \in \Gamma(v, r), \\ \qquad\qquad \gamma_e^f(r^f) = c_{v', \gamma_e(r)} \end{cases}$
- $S^f = \bigcup_{v \in V}\{A_v^f\}$ *with* $A_v^f = \bigcup_{r \in reach(v) \cap A_v} \Gamma(v, r)$

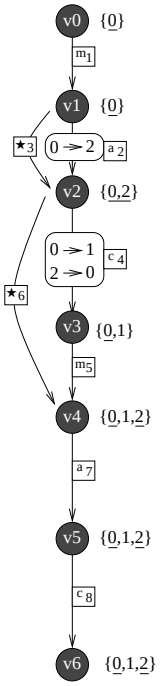


Figure 4: Erasing

The states are now the values $c_{v,r}$, the initial state of $\mathcal{I}^f$ is the default representation of the initial state of $\mathcal{I}$. The transition functions are the identity for edges of the free set. Otherwise, they map a state r in node v to the default value $c_{v', \gamma_e(r)}$. The final states of $\mathcal{I}^f$ are the union of the representations of the reachable final states of nodes of $\mathcal{I}$.

Erasing the minimal instrumented graph of Figure 3 is presented in Figure 4. The chosen free set of edges is $\{m_1, \star_3, m_5, \star_6, a_7, c_8\}$. Only two transitions remain on the resulting instrumentation (compared to six before).

Erasing does not change the semantics of an instrumented graph.

PROPERTY **4** *For all instrumented graph $\mathcal{I}$, $\mathcal{T}(\mathcal{I}) = \mathcal{T}(\mathcal{I}^f)$*

This transformation can be applied to optimize any instrumented graph. In our approach, we apply it after minimization of the direct instrumentation; the result of our transformation chain is then $\mathcal{I}^{d^{m^f}}$.

We pointed out before that minimization improves erasing. In fact, it entails a stronger result.

PROPERTY **5** *For all instrumented graphs $\mathcal{I}$ and $\mathcal{I}^o$,*

$$\mathcal{T}(\mathcal{I}) = \mathcal{T}(\mathcal{I}^o)\quad \Rightarrow\quad \{e\ \mid\ \gamma_e^o = Id\}\ \textit{is a free set of}\ \mathcal{I}^m$$

This property is an optimality result. If $\mathcal{I}^o$ is an optimal instrumentation (i.e. it maximizes the number of identity transitions) equivalent to $\mathcal{I}$, Property 5 tells us that it can be obtained by choosing a maximal free set of the minimization of $\mathcal{I}$. Unfortunately, computing maximal free sets is an NP-complete problem (3-SAT can be encoded into it). An effective heuristics is to pick any spanning tree of the graph. Such a set of edges is guaranteed to be a free set (there

is no sharing and the condition of definition 3 holds trivially). This set can then be enlarged by adding, one by one, edges and checking if the set remains free. The process stops when no edge can be added without loosing freeness. For a structured imperative language, the spanning tree alone ensures that inserting one assignment at each `if` and `while` statement suffices.

3.5 Space optimization

A key step in the erasing transformation is the choice of default values $c_{v,r}$ for the function Γ. The choice made in the previous section is safe but may make $\mathcal{I}^f$ have more states than necessary. This step is formalized by equation (3). A bad choice of these default values may entail that no function Γ respecting the constraints (1) and (2) exists. In order to ensure that a solution exists, the propagation of the default values during the computation of Γ should not lead to the violation of constraint (2). A necessary and sufficient condition is:

$$\forall r_1 \in reach(v_1), \forall r_2 \in reach(v_2), \forall v_1 \xrightarrow{w_1} v,\ v_2 \xrightarrow{w_2} v\ \in F^*,$$
$$\gamma_{w_1}(r_1) \neq \gamma_{w_2}(r_2)\ \Rightarrow\ c_{v_1,r_1} \neq c_{v_2,r_2}$$

Finding default values respecting this condition amounts to coloring an undirected graph whose set of nodes is

$$\{(v, r) \mid v \in V,\ r \in reach(v)\}$$

and where two nodes (v_1, r_1) and (v_2, r_2) are connected if:

$$\exists v \in V,\ v_1 \xrightarrow{w_1} v\ , v_2 \xrightarrow{w_2} v \in F^*\ \wedge\ \gamma_{w_1}(r_1) \neq \gamma_{w_2}(r_2)$$

This graph can be computed by a fix-point iteration. Figure 5 presents such a graph whose coloration is used by the $\mathcal{I}$-graph of Figure 4. Compared to the simple choice presented in Section 3.4, finding a good coloring[3] reduces the number of states of the instrumented graph dramatically. For transformed programs, it means that less space is needed to store transition tables.

4 Context-free graphs

Modeling procedure calls and returns by standard edges is a crude approximation of the actual traces. In order to represent faithfully the control flow of realistic languages, we introduce the notion of *context-free graphs* or *cf-graphs*.

We describe how the steps of the preceding section are adapted to take into account this more general form of graph.

Program abstract representation. A cf-graph $\mathcal{G}_{cf} = (V, v_0, E, Ret, C)$ is a graph along with a new set of nodes ($Ret \subset V$) representing returns and a new set of arrows ($C \subset V \times V \times V$) representing calls. A call edge (v, f, v') is written $v \overset{f}{\hookrightarrow} v'$, and generates no event. Intuitively, it signifies that a program point v calls f and proceeds at v' after f has returned. Return nodes (written *ret* below) do not have outgoing edges. They must be matched with their corresponding call.

In order to define the set of partial traces of cf-graphs, we introduce *return stacks* that are lists of nodes $v_1 : \ldots : v_n : \epsilon$.

[3] The problem if determining the optimal coloring for an arbitrary graph is an NP-complete problem. For control-flow graphs of structured programs the problem becomes polynomial.

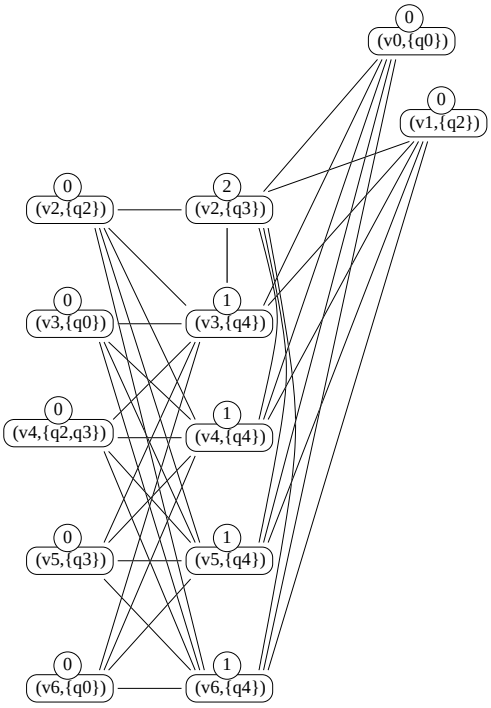


Figure 5: Graph coloring

The path relation is now written $\rightsquigarrow$ and is relative to stacks. For example, $(v, s) \overset{w}{\rightsquigarrow} (v', s')$ means that the path w is a valid execution path starting from node v with stack s and reaching node v' with stack s'. The path relation is formally defined by the rules:

$$(v, s) \overset{\epsilon}{\rightsquigarrow} (v, s)$$

$$\frac{(v, s) \overset{w}{\rightsquigarrow} (v', s') \quad v' \overset{e}{\rightarrow} v''}{(v, s) \overset{we}{\rightsquigarrow} (v'', s')}$$

$$\frac{(v, s) \overset{w}{\rightsquigarrow} (v', s') \quad v' \overset{f}{\hookrightarrow} v''}{(v, s) \overset{w}{\rightsquigarrow} (f, v'' : s')}$$

$$\frac{(v, s) \overset{w}{\rightsquigarrow} (ret, v' : s')}{(v, s) \overset{w}{\rightsquigarrow} (v', s')}$$

The set of all partial traces generated by a cf-graph $\mathcal{G}_{cf}$ is defined as

$$\mathcal{T}(\mathcal{G}_{cf}) = \{w \mid \exists (v, s) \in V \times Stack,\ (v_0, \epsilon) \overset{w}{\rightsquigarrow} (v, s)\}$$

Direct instrumentation. The direct instrumentation is defined exactly as before. An instrumented cf-graph involves a cf-graph instead of a simple graph and call-edges do not bear transitions.

Minimization. A state r is reachable at node v with stack s if there exists an execution starting from node v_0, state r_0 and the empty stack which reaches node v with stack s and state r. This notion of reachability is formalized by the following definition.

$$s\text{-}reach(v, s) = \{\gamma_w(r_0) \mid (v_0, \epsilon) \overset{w}{\rightsquigarrow} (v, s)\}$$

In order to remove useless checks, we need to compute the function *reach* which yields the set of reachable states for each node. This function is defined as the union of the s-reachable states for all possible stacks.

$$reach(v) = \bigcup_{s \in V^*} s\text{-}reach(v, s)$$

This definition does not hint at any constructive way of computing the reachable states. A possible algorithm is to compute first, for each node v and state r, the set of returnable states $\mathcal{R}(v, r)$ defined by:

$$\mathcal{R}(v, r) = \{(ret, \gamma_w(r)) \mid (v, \epsilon) \overset{w}{\rightsquigarrow} (ret, \epsilon)\}$$

A pair of a return node ret and a state r' is returnable from node v and state r if there exists an execution, starting from node v and the empty stack, and reaching the return node ret with an empty stack, which maps r to r'. Those sets are computed by a fix-point iteration finding the smallest solution of:

$$(ret, r) \in \mathcal{R}(ret, r) \qquad \frac{v \overset{e}{\rightarrow} v' \quad (ret, r') \in \mathcal{R}(v', \gamma_e(r))}{(ret, r') \in \mathcal{R}(v, r)}$$

$$\frac{v \overset{f}{\hookrightarrow} v' \quad (ret', r') \in \mathcal{R}(f, r) \quad (ret, r'') \in \mathcal{R}(v', r')}{(ret, r'') \in \mathcal{R}(v, r)}$$

Then, $reach(v)$ can be computed by solving the following constraints (again, this can be done by a fix-point iteration):

- $r_0 \in reach(v_0)$
- $\forall v \overset{e}{\rightarrow} v',\ \gamma_e(reach(v)) \subseteq reach(v')$
- $\forall v \overset{f}{\hookrightarrow} v',\ reach(v) \subseteq reach(f)$
- $\forall v \overset{f}{\hookrightarrow} v',\ \forall r \in reach(v),$ $(ret, r') \in \mathcal{R}(f, r) \quad \Rightarrow \quad r' \in reach(v')$

Concerning the minimization itself, the problem becomes more involved. There is no notion of equivalence relation between states anymore. Even if fusion of states can be defined for instrumented cf-graphs, it may entail changes elsewhere and increase the overall number of states. The approach loses its optimality property as one cannot ensure that the minimized instrumented cf-graph is the best one for erasing.

A pragmatic and effective solution is to "partially" minimize the instrumentation. The previous minimization algorithm is applied to the underlying regular $\mathcal{I}$-graph (i.e. where call edges are interpreted as standard edges, and returns nodes have outgoing edges to all the nodes they may return to). This can be seen as a context insensitive minimization. Then, we proceed with the minimized instrumented cf-graph.

Erasing. The definition of a free set must be extended to take into account the context-free features of instrumentations.

DEFINITION **6** *A set $F \subseteq E$ is said to be free for the instrumented cf-graph $\mathcal{I} = (\mathcal{G}_{cf}, R, r_0, \gamma, S)$ if:*

$$\forall\ (v, s_1) \overset{w_1}{\rightsquigarrow} (v', s_1'),\ (v, s_2) \overset{w_2}{\rightsquigarrow} (v', s_2') \in F^*,$$
$$r \in s\text{-}reach(v, s_1) \cap s\text{-}reach(v, s_2) \ \Rightarrow\ \gamma_{w_1}(r) = \gamma_{w_2}(r)$$

The definition is similar to definition 3 except that both paths (from v to v') may start and end with different stacks.

As before, every edge belonging to a free set may have its transition erased. The erasing phase is almost identical to the previous case. It suffices to add two new constraints to the definition of Γ:

- $\forall v \overset{f}{\hookrightarrow} v',\ \forall r \in reach(v), \quad \Gamma(v, r) \subseteq \Gamma(f, r)$
- $\forall v \overset{f}{\hookrightarrow} v',\ \forall r \in reach(v),\ \forall (ret, r') \in \mathcal{R}(v, r),$ $\Gamma(ret, r') \subseteq \Gamma(v', r')$

Those constraints correspond to the constraint (1) for respectively calls and returns.

Finding an optimal free set is still an NP-complete problem. In practice, we use a heuristic similar to the one previously mentioned. We first take a spanning tree of the underlying regular $\mathcal{I}$-graph (as for minimization) as a free set. We then add edges as long as the set remains free. This step supposes to be able to decide whether a set is free or not (definition 6 does not help much here). In Appendix B, we present how to compute the relation $(v, r) \overset{F}{\rightsquigarrow} (v', r')$ defined by:

$$(v, r) \overset{F}{\rightsquigarrow} (v', r')$$
$$\Updownarrow$$
$$\exists s, s', w \in F^*,$$
$$r \in s\text{-}reach(v, s) \wedge (v, s) \overset{w}{\rightsquigarrow} (v', s') \wedge \gamma_w(r) = r'$$

Two pairs (node,state) are related by $\overset{F}{\rightsquigarrow}$ if it is possible, starting from the first node with the first state, and following only edges of the set F, to reach the second node with the second state. A set F is free iff:

$$\forall v, r, v', r', r'', \quad \begin{array}{c} (v, r) \overset{F}{\rightsquigarrow} (v', r') \\ \wedge \\ (v, r) \overset{F}{\rightsquigarrow} (v', r'') \end{array} \quad \Rightarrow r' = r''$$

As the formula is quantified over a finite domain, it can be tabulated. This provides an algorithm to decide whether a set is free.

5 Abstraction and Concretization

We now sketch the properties that the abstraction and the concretization have to verify in order to ensure the correctness of the approach. In section 5.1 we concentrate on control-flow abstraction and concretization. The techniques described in Section 3 apply to other abstractions as well. We mention in Section 5.2 how we might work on call graphs instead of control-flow graphs.

5.1 Control-flow graphs and trace properties

We suppose that the programming language comes with a deterministic small step operational semantics and that basic instructions are numbered (we will refer to instructions by their integer label). The SOS rules are of the form:

$$(P, \sigma) \overset{k}{\Rightarrow} (P', \sigma') \qquad \text{or} \qquad (P, \sigma) \overset{k}{\Rightarrow} \sigma'$$

where the integer k denotes the instruction reduced by the one-step rewriting. The functional operational semantics is defined by:

$$SOS[\![P]\!]\sigma = \begin{cases} \sigma' & \text{if } (P, \sigma) \overset{w}{\Rightarrow}{}^* \sigma' \\ \textbf{undef} & \text{otherwise} \end{cases}$$

where $\stackrel{w}{\Rightarrow}^*$ denotes the transitive closure of $\Rightarrow$ and w the execution trace as a string of integers.

The annotation phase (see Section 2) has produced a function $\mathcal{E} : \mathbb{N} \to \Sigma$ which maps basic instructions to events. Using this function, a program is abstracted into its control-flow graph $\mathcal{G} = Abs(P, \mathcal{E})$ with annotated events on edges. The safety of the abstraction is expressed by

$$(P, \sigma) \stackrel{w}{\Rightarrow}^* x \implies w \in \overline{\overline{\mathcal{T}(\mathcal{G})}} \tag{4}$$

where $\overline{\overline{\mathcal{T}(\mathcal{G})}}$ is the set of the partial traces of $\mathcal{G}$ reduced to strings of integers.

In the following, we write $(P, \sigma) \stackrel{\epsilon}{\Rightarrow}^* \bot$ to denote aborted computations, and suppose that instructions inserted by the concretization do not produce events (or produce the empty event ϵ).

We define the concretization as a function *Con* which takes a program and an instrumentation and generates a new program. The function *Con* must verify:

$$\begin{array}{l} (P, \sigma) \stackrel{k}{\Rightarrow} (P', \sigma') \ \wedge \ v \stackrel{e}{\longrightarrow} v' \in E \ \wedge e = \mathcal{E}(k)_k \\ \Downarrow \\ \forall q \in reach(v), \\ \textbf{if} \quad \gamma_e(q) \in A_{v'} \quad \textbf{then} \\ \quad (Con(\mathcal{I}, P), st(\sigma, q)) \stackrel{k}{\Rightarrow}^* (Con(\mathcal{I}, P'), st(\sigma', \gamma_e(q))) \\ \textbf{else} \quad (Con(\mathcal{I}, P), st(\sigma, q)) \stackrel{\epsilon}{\Rightarrow}^* \bot \end{array} \tag{5}$$

The function *st* serves to store the state q of $\mathcal{I}$ in σ, γ is the transition function of $\mathcal{I}$, and E the set of edges of $\mathcal{I}$. Intuitively, this condition states that *Con* encodes the instrumentation, but also that the state q can evolve safely (i.e. the source program does not change the state and the introduction of the state does not affect the normal execution). *Con* should satisfy the same constraint for rules of the form $(P, \sigma) \stackrel{k}{\Rightarrow} \sigma'$.

In order to relate source and transformed programs, we define the SOS enforcing the automaton $\mathcal{A}$ as:

$$\frac{(P, \sigma) \stackrel{w}{\Rightarrow}^* x}{(P, \sigma) \stackrel{w}{\Rightarrow}^{\mathcal{A}} x'} \quad \text{where}^4 \quad x' = \begin{cases} x & \text{if } \mathcal{E}(w) \in \mathcal{L}(\mathcal{A}) \\ \bot & \text{otherwise} \end{cases}$$

We can now express the global correctness property.

PROPERTY **7** *If $\mathcal{G}$ is a safe abstraction of P, $\mathcal{I}$ an instrumentation of $\mathcal{G}$ enforcing the automaton $\mathcal{A}$, and Con a concretization that satisfies (5), then:*

$$\forall \ \sigma, \ \ SOS[\![Con(\mathcal{I}, P)]\!] st(\sigma, r_0) =_{st} SOS^{\mathcal{A}}[\![P]\!]\sigma$$

An execution of the transformed program is equivalent to the execution of the original program in parallel with the automaton. Here, the notation $=_{st}$ stands for the equality of stores, regardless of the variable encoding the automaton state.

As an illustration, the concretization function for the toy imperative language used in Figure 1 can be defined as:

$$Con(\mathcal{I}, P) = P[C_1(k)/k]_{k \in instructions(P)}$$

$$\text{with} \qquad C_1(k) = \begin{cases} state := [\gamma_e][state]; \\ \texttt{if } [A_{v'}][state] = 0 \texttt{ then abort}; \\ k \end{cases}$$

[4] To simplify the formulation, we suppose that the function $\mathcal{E}$ is extended to words and that the non final states of $\mathcal{A}$ are trap states.

where $e = \mathcal{E}(k)_k$, $v \stackrel{e}{\longrightarrow} v' \in E$, *state* is a fresh variable, and $[\gamma]$, $[A]$ are constant arrays encoding transitions and sets of states.

In fact, $C_1(k)$ does not insert the assignment if $\gamma_e = Id$. Similarly, if $\gamma_e(reach(v) \cap A_v) \subseteq A_{v'}$ the conditional is not inserted. The concretization *Con* can be shown to satisfy condition (5) (with $st(\sigma, q)$ being implemented by assignments to *state*) and correctness follows.

5.2 Call graphs and stack properties

The core of the approach (Section 3) can be applied to other abstractions and classes of properties. In particular, our technique can be used to enforce security properties defined as regular expressions over stacks. For example, this permits to express the Java (JDK 1.2) security model that is based on stack inspection. In this security mechanism, each time a critical operation is about to be performed, the programmer may request (using a special instruction) a runtime stack inspection. It checks that the permissions of the methods currently present in the return stack suffice to execute the critical operation. The constraint enforced by this inspection can be modeled as a regular expression on return stacks.

Control-flow graphs and traces cannot be used to enforce such security policies. In general, regular expressions on stacks are not regular expressions on traces. In order to use our technique for enforcing this kind of properties, it suffices to change the abstraction and concretization phases. We replace control flow graphs by call graphs. The previous safety property was: "to any partial trace corresponds a path in the graph". It becomes now: "to any stack corresponds a path in the graph".

Concretization differs: it involves passing the state of the instrumentation as a new argument to every function. This technique has already been studied and proposed under the name *security passing style* [23]. The extra argument can be seen as a way of storing the current state of the instrumentation in the current frame. A method (function, procedure) call, corresponds to an edge in the call graph. Just before calling the method or when entering the method (a choice referred as caller-says vs. callee-says [22]), the new automaton state is computed as the image of the previous one by the transition corresponding to this edge. When the execution returns from a method, the frame is popped, and the previous frame (and thus state) is fetched. The store contains one automaton state per frame.

As before, our approach still optimizes the generated code by removing transitions and tests. There are also optimizations specific to call graphs: the code can be optimized further by removing useless extra parameters (technically, if $reach(v)$ is a singleton in the minimized instrumentation).

6 Related Work

We focus in this section on the different approaches for enforcing properties. They can be classified in three groups: the static, dynamic, and mixed approaches. We review them in turn. We also indicate how each approach solves the problem of enforcing a specific and standard property: type safety. This is illuminating since these approaches to type-safety are well-known and they have the same respective advantages and drawbacks as the general approaches.

- *Dynamic approaches.* These approaches often rely on a process running concurrently with the program. The process observes the execution and halts the program whenever it is about to violate the property. Monitors (such as VeriSoft [7] and AMOS [3]) or "security kernels" (such as Schneider's security automata [19]) belong to this class.

  The programming language counterpart of these system approaches amounts to integrating, without any optimizations, the monitor within the program. The enforcement is formally expressed in a single framework (the programming language) by program transformation. This approach makes code more portable and avoids extending the language semantics to security mechanisms. This is also likely to simplify correctness proofs. Our direct instrumentation (step 4 in Section 2) can be seen as a dynamic, programming language, approach. Several other systems use such techniques; for example, Naccio [6] modifies programs to call wrapper functions in order to enforce safety policies.

  These approaches are flexible in that they never reject valid programs and permit to express customized properties. Actually, the enforcement mechanisms being purely dynamic, they may consider a large class of properties (e.g. a monitor may inspect and check the runtime store and stack). The main drawback is the runtime cost. The enforcement process is not specialized to the program and each program instruction may involve a runtime check.

  For type safety, an example of a dynamic approach is the implementation of languages such as Scheme. The runtime type checks involved make the approach flexible but costly.

- *Static approaches.* Within this approach, the programmer is responsible for the enforcement. A static analyzer is then used to check and certify that the property has been properly enforced. To stay with security properties, let us cite the static techniques based on abstract interpretations [15], model-checking [10], or type systems ([9],[20]). The main advantage of these approaches is that they entail no runtime cost. However, they may reject perfectly valid programs and do not address the expressivity problem. In languages such as Java, where code is loaded dynamically, runtime checks are essential. Programmers have to make their code secure beforehand (e.g. by inserting `AccessController.checkPermission` calls [8] throughout the code) so that the program can be statically checked.

  Statically typed languages can be seen as a static approach for enforcing type safety. For this property, the programmer expresses naturally the enforcement by following a programming discipline.

- *Mixed approaches.* Static approaches may reject valid programs and dynamic ones may entail a prohibitive runtime overhead. A mixed strategy does not reject programs (it adds dynamic checks) and aims at minimizing the runtime penalty (using static analyses). Our approach belongs to that class. The recent work by Erlingsson and Schneider [5] considers also the problem of specializing a security automaton to a program. Walker uses their technique and generates secure, certified code in a type-theoretic framework [21]. Compared to our approach, they consider possibly infinite state automata but simpler, local, optimizations. By using global analyses and transformations, we are able to suppress more unnecessary checks and transitions. Also, by working on program abstractions, our framework is more general than the restriction to finite state automata may lead one to think (as hinted at in Section 5.2).

  Soft typing (see e.g. [2]) is a mixed approach for enforcing type safety. A soft type checker does not reject statically ill-typed programs but inserts dynamic type checks where necessary. Its goal is similar to ours: retaining flexibility for the minimum runtime cost.

There are many other works about the enforcement of more specific security properties. Let us cite, for example, approaches which consider runtime checks to ensure type-based security properties ([17], [14]). Even if they focus on a specific property, the work of Wallach and Felten [23] has several common points with ours. They express the security model of Java by a pushdown automaton that is implemented by program transformation. The resulting code is then optimized using a kind of dead code elimination [22]. This is another evidence that, beyond correctness and portability benefits, a programming language approach also permits to specialize/optimize the enforcement with respect to programs.

7 Conclusion

The initial inspiration of our work came from the study of aspect-oriented programming (AOP) [11]. The goal of AOP is to isolate aspects which cross-cut the program basic functionality and whose implementation would otherwise yield tangled code. The transformation process that integrates the aspect into the program is called weaving. Even if AOP is a quite recent concept and still lacks a theory, several convincing case studies suggest that it has great potential [12]. In these case studies, aspects are annotations guiding optimizations, describing the representation of data, or the coordination of threads. Our approach can be seen as an instance of AOP where an aspect is a formal trace property. Like AOP, this separation of concerns leads to programs that are easier to develop and maintain. This feature is especially important in a security context where it is impossible to foresee all possible attacks and where programs may have to be changed quickly to respond to new threats. Furthermore, considering aspects as properties permits to describe and control precisely the semantic impact of weaving (another essential feature for security critical systems).

The material presented in Section 3 (i.e. the intraprocedural case) has been implemented in O'Caml. Simple heuristics have been implemented for the costly steps (i.e. the choice of a free set (Section 3.4) and the graph coloring (Section 3.5)) which make the complexity of the whole process linear in the size of the program. We performed a few experiments on dummy Pascal-like programs and the preliminary results are encouraging. But real examples would make benchmarking much more valuable. As suggested in the introduction, an interesting application of our technique would the securization of mobile code upon receipt. Our prototype could be applied directly if the language of applets were e.g. the simple imperative language of Figure 1.

Dealing with Java byte code supposes several extensions. For example, the abstraction phase must include a control flow analyzer to produce precise graphs. The approach must also take into account the fact that an applet may dynamically load new code. It remains to be seen to which extent this feature prevents optimizations.

There are other avenues for further research. For the time being, we have only considered transformations that stop the program when a violation is detected. More sophisticated forms of error handling could be considered. However, this must be tackled with care in order to keep the semantic impact of transformations under control. A longer term research is the design of a general framework for "property oriented programming" integrating static analyses with program transformations.

Acknowledgments

Thanks are due to Thomas Jensen for his proofreading, to Daniel Le Métayer for commenting on an earlier version of this paper, and to Yoann Padioleau for building a prototype implementation.

References

[1] D. F. Brewer and M. J. Nash. The Chinese wall security policy. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, pages 206–214, May 1989.

[2] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of PLDI '91, Conference on Programming Language Design and Implementation (Toronto, Canada)*, pages 278–292, June 1991.

[3] D. Cohen, M. S. Feather, K. Narayanaswamy, and S. S. Fickas. Automatic monitoring of software requirements. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 602–603, Berlin - Heidelberg - New York, May 1997. Springer.

[4] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B*, pages 995–1072. Elsevier, 1990.

[5] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Sept. 1999.

[6] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Research in Security and Privacy, pages 32–45, Oakland, CA, May 1999. IEEE Computer Society Press.

[7] P. Godefroid. Model checking for programming languages using VeriSoft. In *Conference record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, Paris, 1997. ACM Press.

[8] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems Proceedings, Monterey, California, December 8–11, 1997*, pages 103–112, Berkeley, CA, USA, 1997.

[9] N. Heintze and J. G. Riecke. The SLam calculus: Programming with security and integrity. In *Conference record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, 19–21 Jan. 1998.

[10] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security policies. In *Proc. of Symp. on Research in Security and Privacy*, pages 89–103, May 1999.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of the European Conference on Object-Oriented Programming*, June 1997.

[12] G. Kiczales *et al.* Aspect-oriented programming. Collection of Tech. Reports SPL-97-007 – 010, Xerox Palo Alto Research Center, 1997.

[13] N. Klarlund and M. I. Schwartzbach. A domain-specific language for regular sets of strings and trees. In *Proceedings of Domain Specific Languages*, pages 145–156, 1997.

[14] X. Leroy and F. Rouaix. Security properties of typed applets. In *Conference record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 391–403, 19–21 Jan. 1998.

[15] M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Comp.*, 4(6A):727–754, 1992.

[16] G. C. Necula. Proof-carrying code. In *Conference record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Jan. 1997.

[17] J. Palsberg and P. Ørbæk. Trust in the lambda calculus. In *Proc. of the 1995 Static Analysis Symposium*, volume 983 of *LNCS*, pages 314–330, 1995.

[18] J. Rushby. Kernels for safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, pages 310–320. Blackwell Scientific, 1987.

[19] F. B. Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, 1998. (Revised version July 1999).

[20] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. 7th TAPSOFT*, volume 1214 of *LNCS*, pages 607–621, 1997.

[21] D. Walker. A type system for expressive security policies. In *Conference record of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston, 2000. ACM Press.

[22] D. S. Wallach. *A New Approach to Mobile Code Security.* PhD thesis, Faculty of Princeton University, January 1999.

[23] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *IEEE Symposium on Security and Privacy*, May 1998.

Appendix A

We sketch in this appendix the proofs of the properties stated in the paper.

Let us first introduce the notion of $\mathcal{I}$-graph morphisms.

DEFINITION **8** *Let $\mathcal{I}^1 = (\mathcal{G}, R^1, r_0^1, \gamma^1, S^1)$ and $\mathcal{I}^2 = (\mathcal{G}, R^2, r_0^2, \gamma^2, S^2)$ be two $\mathcal{I}$-graphs. A $\mathcal{I}$-graph morphism $\mathcal{F}$ is a function from $V \times R^1$ to R^2 verifying:*

$$\mathcal{F}_{v_0}(r_0^1) = r_0^2 \tag{6}$$

$$\forall v \xrightarrow{e} v' \in E,\ \forall r \in reach^1(v),\ \mathcal{F}_{v'}(\gamma_e^1(r))) = \gamma_e^2(\mathcal{F}_v(r)) \tag{7}$$

$$\forall v \in V,\ \forall r \in reach^1(v),\ r \in A_v^1 \Leftrightarrow \mathcal{F}_v(r) \in A_v^2 \tag{8}$$

We will write $\mathcal{F}(\mathcal{I}^1) = \mathcal{I}^2$.

The key property of a morphism is to keep the set of partial traces unchanged.

LEMMA **1** $\mathcal{T}(\mathcal{F}(\mathcal{I})) = \mathcal{T}(\mathcal{I})$

PROOF:
Let $\mathcal{I} = (\mathcal{G}, R, r_0, \gamma, S)$ and $\mathcal{F}(\mathcal{I}) = (\mathcal{G}, R^0, r_0^0, \gamma^0, S^0)$. One proves by induction on w using properties (6) and (7), that:

$$\forall\ v_0 \xrightarrow{w} v \in E^*,\ \gamma_w^0(r_0^0) = \mathcal{F}_v(\gamma_w(r_0))$$

The lemma is then a consequence of (8). □

PROOF OF PROPERTY 2:
Let us first note that

$$\forall v \xrightarrow{e} v' \in E,\ \forall r \in reach(v), \gamma_e^m(class_{\mathcal{I}}^v(r)) = class_{\mathcal{I}}^{v'}(\gamma_e(r))$$

defines γ as a function. It is easy to show that the function $v, r \mapsto class_{\mathcal{I}}^v(r)$ is a morphism from $\mathcal{I}$ to $\mathcal{I}^m$. Then, lemma 1 entails the correctness of minimization. □

PROOF OF PROPERTY 4:
Let $\mathcal{F}$ be defined as:

$$\mathcal{F}:\ v, r \mapsto r' \text{ such that } r \in \Gamma(v, r')$$

This defines a function because of the constraints (2) and (3). Furthermore, the definition of γ^f satisfies:

$$\forall\ v \xrightarrow{e} v',\ r \in reach(v),\ \forall r^f \in \Gamma(v, r),\ \gamma_e^f(r^f) \in \Gamma(v, \gamma_e(r))$$

It is then easy to show that $\mathcal{F}$ is a morphism from $\mathcal{I}^f$ to $\mathcal{I}$ and correctness of erasing follows. □

LEMMA **2** *Let $\mathcal{I}$ and $\mathcal{I}^1$ be two equivalent instrumentations (having the same set of traces) then, there exists a morphism from $\mathcal{I}^1$ to $\mathcal{I}^m$.*

PROOF:
Let $\mathcal{I} = (\mathcal{G}, R, r_0, \gamma, S)$ and $\mathcal{I}^1 = (\mathcal{G}, R^1, r_0^1, \gamma^1, S^1)$ verifying:

$$\forall\ v_0 \xrightarrow{w} v \in \mathcal{T}(\mathcal{G}),\ \gamma_w(r_0) \in A_v \Leftrightarrow \gamma_w^1(r_0^1) \in A_v^1$$

Let $\mathcal{F}$ be defined for $r \in reach^1(v)$ by:

$$\mathcal{F}:\ v, r \mapsto \gamma_w^m(r_0^m) \quad \text{where} \quad v_0 \xrightarrow{w} v \wedge r = \gamma_w^1(r_0^1)$$

Let us first show that this definition makes sense. We have to prove that once v and r are fixed, $\gamma_w^m(r_0^m)$ is a constant (with respect to w). In other words,

$$v_0 \xrightarrow{w} v \wedge v_0 \xrightarrow{w'} v \wedge \gamma_w^1(r_0^1) = \gamma_{w'}^1(r_0^1) \Rightarrow \gamma_w^m(r_0^m) = \gamma_{w'}^m(r_0^m)$$

Let $v_0 \xrightarrow{w} v$ and $v_0 \xrightarrow{w'} v$ be two paths such that $\gamma_w^1(r_0^1) = \gamma_{w'}^1(r_0^1)$, then:

$$\begin{aligned} \mathcal{T}(\mathcal{I}[v, \gamma_w(r_0) \mapsto v_0, r_0]) &= \mathcal{T}(\mathcal{I}^1[v, \gamma_w^1(r_0^1) \mapsto v_0, r_0]) \\ &= \mathcal{T}(\mathcal{I}[v, \gamma_{w'}(r_0) \mapsto v_0, r_0]) \end{aligned}$$

It proves that $\gamma_w(r_0) \overset{\mathcal{I}}{\equiv}_v \gamma_{w'}(r_0)$ which is equivalent to $\gamma_w^m(r_0^m) = \gamma_{w'}^m(r_0^m)$. It is then easy to show that the function $\mathcal{F}$ is a morphism from $\mathcal{I}^1$ to $\mathcal{I}^m$. □

A consequence of this lemma is that the minimization is unique up to isomorphism.

LEMMA **3** *Let E be a a free set for $\mathcal{I}$, if there is a morphism from $\mathcal{I}$ to $\mathcal{I}'$ then E is a free set for $\mathcal{I}'$.*

PROOF:
This can be shown using (7) and the definition of a free set for $\mathcal{I}$. □

PROOF OF PROPERTY 5:
The set $\{e \mid \gamma_e^o = Id\}$ is a free set of $\mathcal{I}^o$. Lemma 2 says that there exists a morphism from $\mathcal{I}^o$ to $\mathcal{I}^m$. Then Lemma 3 suffices to deduce the property. □

PROOF OF PROPERTY 7:
In this proof, we will abuse the notation by using k or w to denote (string of) integers representing instructions as well as their associated (string of) events or annotated events. We first prove by induction on the length of w that:

$$\begin{array}{ccc} (P, \sigma) & \overset{w}{\Rightarrow}^{\mathcal{A}} & (P', \sigma') \\ \Downarrow & & \\ (Con(\mathcal{I}, P), st(\sigma, r_0)) & \overset{w}{\Rightarrow}^{*} & (Con(\mathcal{I}, P'), st(\sigma', \gamma_w(r_0))) \end{array}$$

Let us consider P,σ,P' and σ' such that $(P, \sigma) \overset{w}{\Rightarrow}^{\mathcal{A}} (P', \sigma')$.

- If $w = \epsilon$ then $P = P'$ and $\sigma = \sigma'$ (because all instructions are supposed to generate integers in the original program), and the property holds trivially.

- If $w = w'k$, there exists P'' and σ'' such that $(P, \sigma) \overset{w'}{\Rightarrow}^{*} (P'', \sigma'')$ and $(P'', \sigma'') \overset{k}{\Rightarrow} (P', \sigma')$. We have $w \in \mathcal{T}(\mathcal{G})$ because the abstraction is safe (4). Consequently, there exists two nodes v'' and v' such that $v_0 \xrightarrow{w'} v''$ and $v'' \xrightarrow{k} v'$. By definition of the reachability $\gamma_{w'}(r_0) \in reach(v'')$. Furthermore, we know that $w \in \mathcal{L}(\mathcal{A})$ (because $P, \sigma \overset{w}{\Rightarrow}^{\mathcal{A}} P', \sigma'$) and $w \in \mathcal{T}(\mathcal{G})$, thus (by Definition 1) $\gamma_w(r_0) = \gamma_k(\gamma_{w'}(r_0)) \in A_{v'}$. We may then apply the equation 5 and obtain:

$$\begin{array}{l} (Con(\mathcal{I}, P''), st(\sigma'', \gamma_{w'}(r_0))) \overset{k}{\Rightarrow}^{*} \\ \qquad (Con(\mathcal{I}, P'), st(\sigma', \gamma_w(r_0))) \end{array}$$

$$(v,r) \xrightarrow{F} (v,r) \qquad \frac{(v,r) \xrightarrow{F} (v',r') \quad v' \xrightarrow{e} v'' \quad e \in F}{(v,r) \xrightarrow{F} (v'', \gamma_e(r''))} \qquad \frac{(v,r) \xrightarrow{F} (v',r') \quad v' \xhookrightarrow{f} v'' \quad (f,r') \xrightarrow{F} (ret,r'')}{(v,r) \xrightarrow{F} (v'',r'')}$$

Transitions over balanced paths of F.

$$\frac{(v,r) \xrightarrow{\overset{E}{\rightsquigarrow}(v',r')\overset{F}{\rightsquigarrow}} (v'',r'') \quad (v'',r'') \xrightarrow{F} (v''',r''')}{(v,r) \xrightarrow{\overset{E}{\rightsquigarrow}(v',r')\overset{F}{\rightsquigarrow}} (v''',r''')} \qquad \frac{(v,r) \xrightarrow{E} (v',r') \quad (v',r') \xrightarrow{\overset{E}{\rightsquigarrow}(v'',r'')\overset{F}{\rightsquigarrow}} (v''',r''')}{(v,r) \xrightarrow{\overset{E}{\rightsquigarrow}(v'',r'')\overset{F}{\rightsquigarrow}} (v''',r''')}$$

$$\frac{v \xhookrightarrow{f} v'' \quad (f,r) \xrightarrow{\overset{E}{\rightsquigarrow}(v',r')\overset{F}{\rightsquigarrow}} (ret,r'')}{(v,r) \xrightarrow{\overset{E}{\rightsquigarrow}(v',r')\overset{F}{\rightsquigarrow}} (v'',r'')}$$

Transitions over half-F balanced paths.

$$\frac{(v,r) \xrightarrow{\overset{E}{\rightsquigarrow}(v',r')\overset{F}{\rightsquigarrow}} (v'',r'')}{(v',r') \overset{F}{\rightsquigarrow} (v'',r'')} \qquad \frac{(v,r) \overset{F}{\rightsquigarrow} (v',r') \quad (v',r') \xrightarrow{F} (v'',r'')}{(v,r) \overset{F}{\rightsquigarrow} (v'',r'')} \qquad \frac{(v,r) \overset{F}{\rightsquigarrow} (v',r') \quad v' \xhookrightarrow{f} v''}{(v,r) \overset{F}{\rightsquigarrow} (f,r')}$$

Transitions over ordinary paths of F.

Figure 6: Deciding the freeness in the context-free case.

The induction hypothesis on w' gives:

$$Con(\mathcal{I},P), st(\sigma,r_0) \overset{w'}{\Rightarrow}^* Con(\mathcal{I},P''), st(\sigma'',\gamma_{w'}(r_0))$$

and therefore

$$Con(\mathcal{I},P), st(\sigma,r_0) \overset{w}{\Rightarrow}^* Con(\mathcal{I},P'), st(\sigma',\gamma_w(r_0)))$$

which proves the property.

We return to the proof of Property 7. Let $\sigma_r = SOS^{\mathcal{A}}[\![P]\!]\sigma$, two cases may occur:

- If $\sigma_r = \bot$, then there exists w and k such that $(P,\sigma) \overset{wk^*}{\Rightarrow} (P',\sigma')$ with $w \in \mathcal{L}(\mathcal{A})$ but $wk \notin \mathcal{L}(\mathcal{A})$.

  Let v and v' be such that $v_0 \xrightarrow{w} v$ and $v \xrightarrow{k} v'$.

  Applying the property we just proved, we obtain:

  $$(Con(\mathcal{I},P), st(\sigma,r_0)) \overset{w}{\Rightarrow}^* (Con(\mathcal{I},P'), st(\sigma',\gamma_w(r_0))))$$

  We also know that $\gamma_{wk}(r_0) \notin A_v$ because $wk \notin \mathcal{L}(\mathcal{A})$ (Definition 1). Thus, equation 5 yields:

  $$(Con(\mathcal{I},P'), st(\sigma',\gamma_w(r_0)))) \overset{\epsilon}{\Rightarrow}^* \bot$$

  It means $SOS[\![Con(\mathcal{I},P)]\!]st(\sigma,r_0) = \bot$.

- Similarly, if $\sigma_r \neq \bot$, we have:

  $$SOS[\![Con(\mathcal{I},P)]\!]st(\sigma,r_0) = st(\sigma_r,\gamma_w(r_0))$$

  where w is such that $(P,\sigma) \overset{w}{\Rightarrow}^* \sigma_r$.

□

Appendix B

This appendix describes the system of constraints used for deciding whether a set is free in context-free abstractions.

The goal is to compute the relation $(v,r) \overset{F}{\rightsquigarrow} (v',r')$. It can be done by finding the smallest solution of the system of constraints presented in figure 6.

The first set of rules computes the relation $(v,r) \xrightarrow{F} (v',r')$ which is defined by:

$$(v,r) \xrightarrow{F} (v',r') \iff \exists w \in F^*,\ (v,\epsilon) \overset{w}{\rightsquigarrow} (v',\epsilon) \wedge \gamma_w(r) = r'$$

It corresponds to the existence of a path of edges belonging to F, where all calls are matched with returns (and vice versa), that maps r to r'.

The second set of rules describes transitions over the half-F balanced paths. The formal definition is:

$$\begin{array}{c} (v,r) \xrightarrow{\overset{E}{\rightsquigarrow}(v',r')\overset{F}{\rightsquigarrow}} (v'',r'') \\ \Updownarrow \\ \exists s', \quad \exists (v,\epsilon) \overset{w}{\rightsquigarrow} (v',s') \in E^*, \gamma_w(r) = r' \\ \wedge \quad \exists (v',s') \overset{w'}{\rightsquigarrow} (v'',\epsilon) \in F^*, \gamma_{w'}(r') = r'' \end{array}$$

Finally, the third set of rules describes how to compute the relation $(v,r) \overset{F}{\rightsquigarrow} (v',r')$.

What makes this system work is that if $(v_0,\epsilon) \overset{w}{\rightsquigarrow} (v,s)$ then any postfix path w' of w can be written as $w_1'w_2'$, with:

- Any call occurring in w_1' has a corresponding return in w_1'.
- Any return occurring in w_2' has a corresponding call in w_2'.

In our system of constraints, the definition of half F balanced paths only serves to compute w_1'. Then, the third set of rules is used to concatenate w_1' with w_2'.