Small Witnesses for Abstract Interpretation-based Proofs

Frédéric Besson, Thomas Jensen, and Tiphaine Turpin

IRISA/{Inria, CNRS, Université de Rennes 1}
Campus de Beaulieu, F-35042 Rennes, France

Abstract. Abstract interpretation-based proof carrying code uses post-fixpoints of abstract interpretations to witness that a program respects a safety policy. Some witnesses carry more information than needed and are therefore unnecessarily large. We introduce a notion of size of a witness and propose techniques for reducing the size of such certificates. For distributive analyses, we show that a smallest witness exist and we give an iterative algorithm for computing it. For non-distributive analyes we propose a technique for pruning a witness and illustrate this pruning on a relational, polyhedra-based analysis. Finally, only the existence of a witness is needed to assure the code consumer of the safety of a given program. This makes possible a compression technique of witnesses where only part of a witness is sent together with an encoding of the iterative steps necessary to prove that it is part of a post-fixpoint.

1 Introduction

Proof-carrying code (PCC) is a software security infrastructure in which programs come equipped with certificates that allow a code consumer to check that the program respects a given safety policy. There are several requirements to the structure of such certificates which at the same time must be easy to produce for the code producer, small relative to the code size, and simple to check by the code consumer. Initial PCC works used as certificates a lambda-term encoding of proofs [Nec97] to be type-checked by the Logical Framework (LF). To optimise the size of these proofs, Necula and Lee proposed LF$_i$ a compressed proof format for LF terms [NL98]. For a weaker logic, Necula and Rahul transmit as certificate an oracle (a stream of bits) that guides a higher-order logic interpreter in its proof search [NR01]. Wu, Appel, Stump [WAS03] show how to combine these ideas with dedicated program logics in order to obtain foundational proof checkers with small witnesses. Albert *et al.,* [AAPH06] propose abstract interpretation as a way to fully automate the generation of certificates. In this approach, fixpoints (invariants) play the role of certificates and a checker will have to verify *a)* that a proposed certificate is indeed a fixpoint of an abstract interpretation of the program and *b)* that this fixpoint entails the safety policy.

An important issue is how to encode such certificates in a manner that keeps the certificate small while still allowing efficient checking. This paper propose a theory based on abstract interpretation for studying this issue. Given an abstract interpretation F of program p over an abstract domain D of program properties, and a safety policy expressed as a property ϕ in D, we study the set of *witnesses* of ϕ with respect to p, *i.e.*, the elements $w \in D$ satisfying $F(w) \sqsubseteq w \sqcap \phi$. Section 2 formalises the notion of witness and discuss how to optimise the *size* of a witness.

For certain kinds of abstract interpretations it is possible to guarantee the existence of a smallest witness for any abstract property. This is the case *e.g.*, for distributive data flow analyses [MJ81,MR90] and disjunctive-complete abstract interpretations. In Section 3 we provide a fixpoint characterisation of the smallest witness of a given abstract property for any distributive analysis and illustrate how to obtain an effective algorithm for a class of set-based analyses. In the general case, it is impossible to compute a smallest witness without resorting to an exhaustive search. Instead we propose in Section 4 a technique for *pruning* a witness to obtain a witness that is smaller relative to the initial witness. We illustrate this by showing how to prune the result of a relational, polyhedra-based abstract interpretation.

For the PCC application of fixpoint compression it is important to note that it is the *existence* of a witness that matters. This makes further optimisations possible because a code certificate now only have to convey the code consumer with sufficient information to convince him that he is able to build a witness. To make this idea concrete, we define in Section 5 certificates as *strategies* that encode the steps that an iterative fixpoint solver will take in order to reconstruct a complete fixpoint given the values at selected program points. This can be seen as a generalisation of the Lighwteight Java Byte Code using stack maps defined by Rose [Ros03] and used in the KVM Java virtual machine for embedded devices. Section 6 discusses related work, notably the recent proposal by Albert *et al.* [AAPH06] for reducing fixpoints produced by a generic fixpoint algorithm.

2 Obtaining witness from abstract interpretation

Central to PCC is the ability to generate checkable proofs of programs. Previous works have shown how to obtain a proof from abstract interpretations. The key insight is that abstract interpretation does not return a yes/no answer but a property which over-approximates the program behaviour. In abstract interpretation terms, the notion of approximation is formalised by a Galois insertion between the semantic (concrete) domain of the program and the abstract domain of properties. A correct over-approximation of the program behaviour is a post-fixpoint of the abstraction of the program. As a result, proving that a program verifies a property, say ϕ, amounts to proving that there exists a post-fixpoint of the abstraction of the program semantics, say ψ, which entails ϕ. Under these conditions, this is a basic result from the theory of abstract interpretation [CC77] that the least fixpoint of the program semantics satisfies the property.

$$(\exists \psi, [\![p]\!]^\sharp(\psi) \sqsubseteq \psi \wedge \psi \sqsubseteq \phi) \Rightarrow \mathrm{lfp}([\![p]\!]) \vDash \phi$$

In a static analysis context, the abstract semantics ($[\![.]\!]^\sharp$) and the ordering of properties ($\sqsubseteq$) are computable functions. Therefore, given the property ψ, checking that a program verifies a property ϕ is a straightforward computation.

2.1 Witnesses

This motivates the definition of a proof witness for abstract interpretation.

Definition 1 (Direct witness). *A* direct *witness for a property $\phi \in D$ and a (monotone) abstract operator $F : D \to D$ is an abstract property $w \in D$ such that w is a post-fixpoint of F ($F(w) \sqsubseteq w$); and w entails ϕ ($w \sqsubseteq \phi$).*

This definition of a witness is the naive instantiation of PCC in the context of abstract interpretation. We propose to study a larger class of witnesses that are compact to encode and as fast to check. The key observation here is that verifying a witness involves some unavoidable computation of F, the results of which need not appear explicitly in the witness. To this end, Definition 2 relaxes the notion of direct witness while preserving its role (the existence of a witness entails the satisfaction of ϕ) and keeping the same verification cost.

Definition 2 (Witness). *An abstract interpretation witness proof for a property $\phi \in D$ and a (monotone) abstract operator $F : D \to D$ is an abstract property $w \in D$ such that $F(w) \sqsubseteq w \sqcap \phi$.*

The following Lemmas affirms that witnesses are as good as direct ones for proving ϕ, and that there are more of them than direct witnesses.

Lemma 1.

1. *If w is a witness then $F(w)$ is a direct witness.*
2. *If w is a direct witness then w is a witness.*

Proof Sketch. Follows directly from the monotonicity of F, the definition of the greatest lower bound operator ($\sqcap$) and the transitivity of the ordering $\sqsubseteq$. □

We focus on optimising the latter, more general version of witnesses.

2.2 On the size of witnesses

When choosing a witness, there are two criteria of interest: its size and its verification cost. In this paper we focus on the size of witnesses, but the results in Sections 3 and 4 should be a good starting point for reducing the verification cost, at least in terms of memory.

In the theory of abstract interpretation, the least fixpoint (lfp(F)) is the strongest property that can be proved of a program and is therefore a poor choice for a witness, because it contains information that is not needed for proving a particular property. *E.g.,* to prove a property at a specific program point (such as absence of array accesses out of bounds or the absence division by zero) only a few program variables and a few program points are relevant. For the others, no information is needed. So, we will rather search for weaker witnesses which are usually smaller because they encode the minimal amount of information needed to prove the property. Notice that the program property to be proved is usually not a witness because it is not a post-fixpoint of F.

To make this argument more precise, consider standard data flow analyses that compute a property for each program point. These analyses operate on a product lattice D^n where n is the number of program points, or even (if we further refine the decomposition) the number of pairs (pp, v) where pp is a program point and v is a variable. Lattice elements are n-tuples for which the ith projection is, for example, a formula characterizing the property of the ith program point. The ordering is point-wise

$$(\psi_1, \ldots, \psi_n) \sqsubseteq (\psi_1', \ldots, \psi_n') \text{ iff } \psi_1 \sqsubseteq \psi_1' \wedge \ldots \wedge \psi_n \sqsubseteq \psi_n'$$

and the size of the property of the whole program is the sum of the size of the atomic formulae.

$$\mid (\psi_1, \ldots, \psi_n) \mid = \mid \psi_1 \mid + \ldots + \mid \psi_n \mid$$

As argued above, for a number of program points these ψ's can be set to $\top$ (and hence left out) because they are not needed for proving the particular property. This suggests that as a general rule, smaller witnesses are those that are weaker (higher up) in the lattice ordering $\sqsubseteq$. While not universally true, this is valid for all analyses based on lattices obtained as meet-completions of sets of unordered atomic properties and, for the present paper, we will adopt the principle that the smaller witnesses are those that are higher in the lattice ordering.

3 Optimal witnesses for distributive analyses

In this section, we show that for distributive analyses it is possible to compute the *weakest* witness which, as soon as our size assumptions are verified, is also the smallest. We also provide an algorithm for computing such optimal witnesses for a class of set-based distributive analyses which includes classical data flow problems such as live variables and reaching definitions [MJ81,MR90].

3.1 Lattice of witnesses

We show that for distributive analyses, witnesses form a lattice. As a consequence, there exists a weakest witness (provided the set of witnesses is not empty). In the following, we consider a lattice of abstract properties D and a distributive function F (i.e., such that $\forall X \neq \emptyset, F\left(\bigsqcup_{x \in X} x\right) = \bigsqcup_{x \in X} F(x)$).

Theorem 1. *Let W be the set of witnesses for a distributive function F and a property ϕ. If W is not empty then $(W, \mathit{lfp}(F), \sqcup, \sqsubseteq)$ is a complete lattice.*

Proof. Because lfp(F) is the least fixpoint of F it is also the least post-fixpoint. As a result, as the set of witnesses is not empty, it is also the least witness.

It remains to show that the least upper bound operators is well-defined *i.e.,* the least upper bounds of witnesses is also a witness: $\forall S \subseteq W, \sqcup S \in W$. By definition of a witness, we have that for all $w \in S$, $F(w) \sqsubseteq w \sqcap \phi$. Since F is distributive, we have that $F(\bigsqcup_{w \in S} w) = \bigsqcup_{w \in S} F(w) \sqsubseteq \bigsqcup_{w \in S} (w \sqcap \phi) = \sqcup S \sqcap \phi$. It follows that $\sqcup S$ is a witness. □

As a result, the weakest witness ww is the least upper bound of all witnesses and is given by ww $= \bigsqcup W$.

3.2 Weakest witnesses as greatest fixpoints

In this section, we show that the weakest witness is the greatest fixpoint of the function $\widetilde{F}$ which given a x computes the weakest precondition wp such that $F(wp) \sqsubseteq x \sqcap \phi$.

Definition 3. *Let F be a distributive function and ϕ a property. $\widetilde{F} : D \to D$ is the function defined by: $\widetilde{F}(x) = \bigsqcup\{y \mid F(y) \sqsubseteq x \sqcap \phi\}$.*

Theorem 2 states that ww, if it exists, is the greatest fixpoint of $\widetilde{F}$.

Theorem 2. *Let F be a distributive function and ϕ be an abstract property. If the greatest fixpoint of $\widetilde{F}$ is not undefined ($gfp(\widetilde{F}) \neq \bot$) then it is the weakest witness of ϕ ($ww = gfp(\widetilde{F})$).*

Proof. We show that the witnesses of ϕ are exactly the pre-fixpoints of $\widetilde{F}$ *i.e.,* $W = \{x \mid x \sqsubseteq \widetilde{F}(x)\}$.

- $\subseteq$: Assume that $w \in W$. By definition of a witness, we have $F(w) \sqsubseteq w \sqcap \phi$. It follows that $w \in \{y \mid F(y) \sqsubseteq w \sqcap \phi\}$. By definition of the least upper-bound operator, we obtain that $w \sqsubseteq \bigsqcup\{y \mid F(y) \sqsubseteq w \sqcap \phi\} = \widetilde{F}(w)$. Therefore, $w \in \{x \mid x \sqsubseteq \widetilde{F}(x)\}$.
- $\supseteq$: Assume that w is a pre-fixpoint of $\widetilde{F}$: $w \sqsubseteq \bigsqcup\{y \mid F(y) \sqsubseteq w \sqcap \phi\}$. By monotony and distributivity of F, we get $F(w) \sqsubseteq F\left(\bigsqcup\{y \mid F(y) \sqsubseteq w \sqcap \phi\}\right) = \bigsqcup\{F(y) \mid F(y) \sqsubseteq w \sqcap \phi\}$. By definition of $\sqcup$, we also have $\bigsqcup\{F(y) \mid F(y) \sqsubseteq w \sqcap \phi\} \sqsubseteq w \sqcap \phi$. By transitivity, we obtain that $F(w) \sqsubseteq w \sqcap \phi$ *i.e.,* $w \in W$.

We conclude, since ww is defined as the greatest witness, that it is the greatest pre-fixpoint of $\widetilde{F}$ and therefore its greatest fixpoint. □

As a result, if the lattice of properties satisfies the *finite descending chain condition*, the weakest witness can be computed by fixpoint iteration: ww $= \widetilde{F}^{\infty}(\top)$.

3.3 Weakest witnesses for set-based analyses

The specification of the function $\widetilde{F}$ is not directly executable. However, for set-based distributive analyses, $\widetilde{F}$ can be derived symbolically without resorting to a naive tabulation. Canonical set-based distributive analyses are data-flow analyses such as available expressions, busy expressions and live variables analyses [MJ81,MR90]. We illustrate the symbolic computation of $\widetilde{F}$ for data flow problems which solution is expressed as the solution of a distributive function F defined component-wise $F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$ such that each F_i is defined by a set expression se of the following form

$$se ::= y \mid c \mid se_1 \cap c \mid se_1 \cup se_2$$

where y is a variable, c is a constant set, $\cap$ is set intersection and $\cup$ is set union.

To compute $\widetilde{F}$ symbolically, the key insight is the definition of a *weakest precondition* operator wp. Given a set expression e and a upper-bound b for this expression, it computes the greatest n-tuple $(v_1, \ldots, v_n)$ so that $e(v_1, \ldots, v_n) \sqsubseteq b$. In other words, it computes the weakest precondition over the variables such that the set expression is dominated by the upper-bound b.

Definition 4. *The weakest precondition operator wp is inductively defined by*

$$\begin{array}{ll} wp(x_j)(b) & = \top^n[j \mapsto b] \\ wp(c)(b) & = \mathit{if}\ c \subseteq b\ \mathit{then}\ \top^n\ \mathit{else}\ \bot^n \\ wp(e \cap c)(b) & = wp(e)(b \cup \bar{c}) \\ wp(e \cup e')(b) & = wp(e)(b) \sqcap wp(e')(b) \end{array}$$

where $\bar{c}$ is the complement of c and $\sqcap$ is point-wise intersection of n-tuples.

Lemma 2 states formally that wp is a weakest precondition operator.

Lemma 2. *Given a set expression e and a set bound b, the following holds:*

$$wp(e)(b) = \bigsqcup\{(x_1, \ldots, x_n) \mid e(x_1, \ldots, x_n) \subseteq b\}.$$

Proof Sketch. The proof is by induction over the set expression e. The cases $e = x_j$ and $e = c$ are proved by definition of wp and $\subseteq$. The remaining cases $e = e_1 \cap c$ and $e = e_1 \cup e_2$ are proved by induction hypothesis using the facts that $e_1 \cap c \subseteq b$ iff $e_1 \subseteq b \cup \bar{c}$ and that $e_1 \cup e_2 \subseteq b$ iff $e_1 \subseteq b \wedge e_2 \subseteq b$. □

Theorem 3 states that $\widetilde{F}$ can be computed using wp.

Theorem 3. *Let F be a function defined by $F(x) = (F_1(x), \ldots, F_n(x))$ and $\phi = (\phi_1, \ldots, \phi_n)$ be a tuple of set properties. We have that the* inverse *of F with respect to ϕ ($\widetilde{F}$) is alternatively defined by:*

$$\widetilde{F}(x) = \bigsqcap_{i \in [1,n]} wp(F_i)(x_i \cap \phi_i).$$

Proof. By definition, we have $\widetilde{F}(x) = \bigsqcup\{y \mid F(y) \sqsubseteq x \sqcap \phi\}$. Because F and ϕ are tuples, this can be rewritten as: $\widetilde{F}(x_1, \ldots, x_n) = \bigsqcup\{y \mid \bigwedge_{i \in [1,n]} F_i(y) \subseteq x_i \sqcap \phi_i\} = \bigsqcup \bigcap_{i \in [1,n]} \{y \mid F_i(y) \subseteq x_i \cap \phi_i\} = \bigsqcap_{i \in [1,n]} \bigsqcup\{y \mid F_i(y) \subseteq x_i \cap \phi_i\}$ (the last equality holds because the sets are downward closed and because the F_i are monotone). By Lemma 2, we finally obtain $\widetilde{F}(x) = \bigsqcap_{i \in [1,n]} wp(F_i)(x_i \cap \phi_i)$. □

4 Fixpoint pruning

In theory, for analyses that are not distributive, it would be possible to make them distributive by disjunctive completion. However, this approach generally leads to analyses of forbidding complexity. In this section, we develop a method for *pruning* a computed (post-)fixpoint into a small witness of a given property, by computing a sort of disjunctive completion relative to the initial post-fixpoint. We will first develop the general pruning technique and then show how its workings for a relational polyhedra analysis.

4.1 General algorithm

Let $w \in D$ be a witness of the property ϕ. Our only assumption is that the property is expressed as a set of constraints (*e.g.,* $\{x + y \geq 0, x \leq 42\}$ for a polyhedral analysis). In that case, we have that the powerset $\mathcal{P}(w)$ of w is a sublattice of D when ordered by set inclusion. The idea of pruning is now simply to look for smaller witnesses in this powerset. For flow-sensitive analyses, the number of constraints is at least proportional to the size of the program and can increase quickly if there are many variables. Therefore, a global minimization of the witness by a direct search in the power set is not feasible. It is however possible to adapt the algorithm from the previous section to minimize a witness.

The disjunctive completion $D^\vee$ of $\mathcal{P}(w)$ is the lattice that contains every disjunction of elements of $\mathcal{P}(w)$. As a unique representation, we choose to represent its elements as sets of maximal disjuncts (sometimes called "crowns" [DP90]).

Lemma 3. *Let F be a function and ϕ a property on a domain D of constraints sets. Let $D^\vee = \{X \subseteq \mathcal{P}(w) \mid \forall x, y \in X, x \sqsubseteq y \implies x = y\}$, and define $X \sqsubseteq^\vee Y$ by $\forall x \in X, \exists y \in Y\ x \sqsubseteq y$. The disjunctive completion $(D^\vee, \sqsubseteq^\vee)$ of $\mathcal{P}(D)$ is a complete lattice whose least upper bound operator satisfies $X \sqcup^\vee Y = \{x \in X \cup Y \mid \forall y \in X \cup Y\ x \sqsubseteq y \implies x = y\}$. Furthermore, letting $F^\vee(X) = \bigsqcup^\vee_{x \in X} \{\bigsqcap\{y \in \mathcal{P}(w) \mid y \sqsupseteq F(x)\}\}$, the existence of witnesses for $F^\vee$ with respect to the property $\{\phi\}$ ensures the safety of the program.*

These are standard results. The second part follows from the existence of a Galois connection between the concrete domain and $D^\vee$ that makes $F^\vee$ an over-approximation of the semantics. □

$F^\vee$ is distributive, so we can use $\widetilde{F^\vee}$ (Definition 3) to compute an optimal witness.

The problem is that this weakest witness is in $D^\vee$ and therefore its minimality doesn't implies that it is small. Intuitively, it contains all possible minimal proofs of the security property and hence can be very large, if there are many disjuncts. We thus take a slightly different way in order to keep witnesses in $\mathcal{P}(w)$. While $\widetilde{F^\vee} : D^\vee \to D^\vee$ is defined by

$$\widetilde{F^\vee}(X) = \bigsqcup^\vee \{Y \in D^\vee \mid F^\vee(Y) \sqsubseteq^\vee X \sqcap^\vee \{\phi\}\}$$

we define on the same lattice $D^\vee$ a variant $\widehat{F}$ whose result is further constrained.

Definition 5. *Let F be a function and ϕ a property on a domain D of constraints sets. The function $\widehat{F} : D^\vee \to D^\vee$ for pruning w is defined by*

$$\widehat{F}(X) = \bigsqcup^\vee \{\{y\} \mid y \in \mathcal{P}(w) \wedge \exists x \in \mathcal{P}(w)\ \{x\} \sqsubseteq^\vee X \wedge F(y) \sqsubseteq x \sqcap \phi \wedge y \sqsubseteq x\}.$$

We remark that $\widehat{F}$ is monotone and let $\widehat{W}$ be its greatest (pre-)fixpoint.

In this definition, the quantification $\exists x \in \mathcal{P}(w)\ \{x\} \sqsubseteq^\vee X\ \wedge \ldots$ can be replaced equivalently by the more direct formula: $\exists x \in X \ldots$. Since $D^\vee$ is of finite height, $\widehat{W}$ can be computed as $\widehat{W} = \widehat{F}^\infty(\top^\vee)$. The following theorem establishes how $\widehat{F}$ and $\widehat{W}$ are used for pruning.

Theorem 4. *$\widehat{W}$ is the set of maximal witnesses in $\mathcal{P}(w)$.*

Proof. We proceed in three steps.

- We first prove that every $w' \in \widehat{W}$ is a witness. As $\widehat{W} = \widehat{F}(\widehat{W})$, by definition of $\widehat{F}$ and the property of $\sqcup^{\vee}$ (Lemma 3), there exists an $x \in \mathcal{P}(w)$ such that $\{x\} \sqsubseteq^{\vee} \widehat{W} \ \wedge\ F(w') \sqsubseteq x \sqcap \phi \ \wedge\ w' \sqsubseteq x$. Since $w' \in \widehat{W}$ we also know that $\{w'\} \sqsubseteq^{\vee} \widehat{W}$ and that w' is maximal with respect to this property. Thus, from $\{x\} \sqsubseteq^{\vee} \widehat{W}$ and $w' \sqsubseteq x$, we deduce that $x = w'$. Therefore, $F(w') \sqsubseteq x \sqcap \phi$, i. e., w' is a witness.
- $\supseteq$: Let w' be a maximal witness. We have that $F(w') \sqsubseteq w' \sqcap \phi \wedge w' \sqsubseteq w'$. So, $\{w'\}$ is a pre-fixpoint of $\widehat{F}$. We conclude by definition of $\widehat{W}$ that $\{w'\} \sqsubseteq^{\vee} \widehat{W}$, that is, $w' \sqsubseteq w''$ for some $w'' \in \widehat{W}$ As shown before, w'' is a witness, therefore $w' = w''$ (because w' is a maximal witness) and $w' \in \widehat{W}$.
- $\subseteq$: We can now finish the proof of the first inclusion. Let $w' \in \widehat{W}$, w' is therefore a witness. Let w'' be a maximal witness greater than w'. From the second inclusion, $w'' \in \widehat{W}$, which implies that $w' = w''$ by definition of $D^{\vee}$ as a set of crowns. Thus w' is a *maximal* witness. □

The actual computation of the greatest fixpoint of $\widehat{F}$ is feasible if the disjunctions have a reasonable number of disjuncts, but this might not always be the case (theoretically, this number can be exponential in the number of constraints). In this case, we can further approximate the optimal solution. We start with the following remark: a disjunction can be under-approximated by any of its disjuncts. Therefore, we can make the pruning feasible by just choosing one disjunct at each step, rather than keeping them all. This leads to the the definition of $\hat{F}$ that is an (non-deterministic) under-approximation of $\widehat{F}$.

Definition 6. *The partial (non-deterministic) function $\hat{F} : \mathcal{P}(D) \to \mathcal{P}(D)$ for approximatively pruning w is defined by*

$$\hat{F}(x) = \textit{choose a weakest}\, y \sqsubseteq x \textit{ s.t. } F(y) \sqsubseteq x \sqcap \phi.$$

It is easy to see that every pre-fixpoint of $\hat{F}$ (formally, every x such that $x \sqsubseteq \hat{F}(x)$ for some choice) is a witness. We get a small one by computing $\hat{F}^{\infty}(\top)$. Also, every optimal witness can be reached by applying the approximated pruning.

4.2 Polyhedra analysis

We illustrate the pruning algorithm on a convex polyhedra analysis. This analysis infers linear invariants that can be used to prove among other properties the absence of integer overflows and illegal array accesses. The domain of convex polyhedra has been used in various contexts, notably to analyse imperative programs [CH78] and synchronous programs [Hal93]. To focus the presentation, we consider the case of linear transition systems.

Definition 7. *A linear transition system is defined by :*

- *a finite set S of locations*
- *a finite set V of integer variables*
- *a finite set E of edges of the form $s \xrightarrow{p} s'$ where $s, s' \in S$ and p is a convex polyhedron of $\mathbb{R}^{V \cup V'}$ (with $V' = \{v' \mid v \in V\}$ a primed copy of V)*
- *a function I that maps every location to a convex polyhedron of $\mathbb{R}^V$.*

The operational semantics is as follows: if there is an edge $s \xrightarrow{p} s'$ then the system performs a transition from the location s with a valuation $\sigma \in \mathbb{R}^V$ of the variables to the location s' with valuation $\tau \in \mathbb{R}^V$ iff $\sigma + \tau' \subseteq p$ where $\sigma + \tau'(v) = \sigma(v)$ and $\sigma + \tau'(v') = \tau(v)$. p can describe assignments and guards with linear expressions. $I(l)$ represents the possible valuations of the initial states whose location is l, and is typically false for all but one location.

The abstract semantics of a linear system is defined over the product lattice $D = \mathrm{Pol}(V)^S$ (where $\mathrm{Pol}(V)$ is the lattice of convex polyhedra of $\mathbb{R}^V$) as the least fixpoint of the function $F : D \to D$ defined by:

$$F(x) = \left[s' \mapsto I(s') \sqcup^{\mathrm{Pol}(V)} \bigsqcup_{s \xrightarrow{p} s' \in E}^{\mathrm{Pol}(V)} [\![p]\!](x(s)) \right]$$

where the abstract semantics $[\![p]\!] : \mathrm{Pol}(V) \to \mathrm{Pol}(V)$ of a particular transition polyhedron p is defined by

$$[\![p]\!](x) = \mathrm{proj}_{V'} \left(x \times \mathbb{R}^{V'} \sqcap^{\mathrm{Pol}(V \cup V')} p \right) [\forall v\ v/v'] \,.$$

Here, $\mathrm{proj}_{V'} : \mathrm{Pol}(V \cup V') \to \mathrm{Pol}(V')$ is the polyhedra projection on the $\mathbb{R}^{V'}$ subspace, and $[\forall v\ v/v']$ is the substitution that "unprimes" every variable.

If we consider that the elements of $\mathrm{Pol}(V)$ are represented as sets of constraints then the whole abstract domain can be defined as a sets of constraints by the coding $x \leftrightarrows \bigcup_{s \in S} \{s\} \times x(s)$.

So, given an abstract property $w \in \mathrm{Pol}(V)^S$ we can compute the weakest precondition operator $\widehat{F} : \widetilde{\mathcal{P}(w)} \to \widetilde{\mathcal{P}(w)}$ as described above. This can be formulated in terms of basics operations on sets and polyhedra.

$$\widehat{F}(X) = \bigsqcup_{\substack{x \in X \\ I \sqsubseteq x}}^{\widetilde{\mathcal{P}(w)}} \bigsqcap_{s \in S}^{\widetilde{\mathcal{P}(w)}} \bigsqcap_{s \xrightarrow{p} s'}^{\widetilde{\mathcal{P}(w)}} \bigsqcup_{\substack{C \subseteq w(s) \\ [\![p]\!](C) \sqsubseteq x(s')}}^{\widetilde{\mathcal{P}(w)}} \{\{s\} \times C\}$$

The meaning of this formula is that, starting from a set of witness candidates, we keep those that are satisfied by the initial condition, compute the set of weakest preconditions in one step for each of them, merge the result and keep only the weakest of the computed properties, i.e., those that do not imply any other such precondition (outermost $\sqcup$). For each candidate, the computation of maximal preconditions can be done state by state (outer $\sqcap$), taking the cross-product : note that this $\sqcap$ can be implemented as a kind of product (a cartesian product where (a, b) is replaced by $a \cup b$) because the terms are independent (they operate on different states). Finally, for every state we take the set of maximal properties that are preconditions of every successor ($\sqcap\sqcup$).

The non-optimal version of pruning can also be applied to polyhedra analysis: instead of keeping a set of maximal witness candidates, we only keep one. The outer $\sqcup$ thus disappears. For every transition, only one weakest precondition of the constraints in its successor state is choosen, removing the innermost $\sqcup$. Therefore, no disjunctions are created anymore, and every gratest lower bound $\{a\} \sqcap \{b\}$ can be replaced by $\{a \cup b\}$. We obtain the simplified partial operator $\hat{F} : D \to D$ with the following inplementation:

$$\hat{F}(x) = \begin{cases} \bigcup_{s\in S} \{s\} \times \bigcup_{s\xrightarrow{p}s'} \begin{array}{l} \textit{choose a weakest } C \subseteq w(s) \\ \textit{s.t. } [\![p]\!](C) \sqsubseteq x(s') \end{array} & \textit{if } I \sqsubseteq x \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

We have tested this algorithm to reduce linear invariants produced by the linear systems analyser StInG [SSM04]. For a given property, we iterate the witness optimisation function $\hat{F}$ until a fixpoint is reached. For the choice function, we use a greedy heuristics which minimises (locally) the constraints to be added to the witness.

As a first example we consider a simple version of bubble sort whose code is shown in Figure 1. We want to prove that array accesses are safe. Therefore, the property is that $0 \leq j \leq |t| - 1$ must hold in the body of the inner loop : $\phi = \{(swap, 0 \leq j), (swap, j \leq |t| - 1)\}$. The program is represented by the linear transition system of Figure 1. The effect of pruning is shown in Table 1. Basically, we find that the upper bound of j is unnecessary to keep because it is implied by the guards. On the other hand, the lower bound can only be proved by induction.

```
for i = 0 to |t| - 2
  for j = 0 to |t| - 2
    exchange t[j] and t[j+1] if needed
```

Fig. 1. Linear transition system for a simple bubble sort

Table 1. Pruning a witness for bubble sort

Location	Initial polyhedron	Remaining constraints
start	$j = 0, \|t\| \geq 0$	$j = 0$
loop	$j \geq 0, \|t\| - j \geq 0$	$j \geq 0$
swap	$j \geq 0, \|t\| - j - 2 \geq 0$	$j \geq 0$

Other examples have been processed in the same way. For instance, for a variant of the classic “train beacon” [Hal98], our witness for proving that trains cannot collide only keeps 7 of the 18 linear invariants generated by StInG. Not all the programs we have tested show a dramatic reduction of the number of constraints. However, these examples are very abstract and only model aspects relevant to the property. For more realistic applications, we expect that more pruning would be possible.

5 Certificates

As stated in the introduction, it is the *existence* of a witness that matters, not its actual content. Based on this observation, we propose to define a *certificate* and an algorithm for checking such certificates such that if the algorithm accepts the certificate then the existence of a witness is guaranteed. We propose a format of certificates, define the algorithm for decoding such certificates and prove the correctness of the algorithm. Then we show how to generate those certificates from a witness, whose reconstruction costs no more than aplying F once, checking $\sqsubseteq$ once and checking ϕ on an abstract property.

Recall that static analyses which attach a property per program point operate over a product lattice D^n where D is the domain of properties and n the number of program points. Note that we could also have a product of different domains, which is equivalent to taking a “sum” lattice for D. The abstract semantics function $F : D^n \rightarrow D^n$ exhibits static dependencies between program points and if we note $x = (x_1, \ldots, x_n)$ then F has the form:

$$F(x) = (F_1(x_{i_{1,1}}, \ldots, x_{i_{1,k_1}}), \ldots, F_n(x_{i_{n,1}}, \ldots, x_{i_{n,k_n}})).$$

Intuitively, properties attached to a particular program point only depends on a subset of the other program points. Typically, for intra-procedural analyses, F_j is only defined with respect to the predecessors of j in the control flow graph. In the following, we write $\Pi_j(x_1, \ldots, x_n) = (x_{i_{j,1}}, \ldots, x_{i_{j,k_j}})$ for the arguments of F_j. In the next section, we propose an algorithm which exploits such dependencies to rebuild a witness from sparse certificates.

5.1 Certificate format and checking algorithm

Existing witness reconstruction algorithms [Ros03,BJP06,AAPH06] are using as certificate a sparse *direct* witness (Definition 1). The current algorithm is more flexible: it relies on a more relaxed definition of witness (see Definition 2) and allows to iterate the F_js more than once. Together, these properties can be exploited to obtain smaller certificates. Definition 8 presents the format of certificates for a product domain D^n.

Definition 8. *A certificate is a pair* (K, S) *where* $K : [1, n] \mapsto D$ *is a* partial *mapping from program points to properties and* $S \in [1, n]^*$ *is a sequence of program points.*

The meaning of a certificate (K, S) is that, starting from an abstract state defined by K (with all undefined program points interpreted as $\top$), and recomputing the program points in S using the F_js should result in a direct witness.

The algorithm for checking a certificate (K, S) is formally defined in Figure 2. We first prove an invariant that entails the correctness and allows for an optimization of the algorithm.

```
check(K, S) =
    check that every $j$ defined in $K$ appears at least once in $S$          (1)
    let $w \in D^n$ be defined as $w = \left[ j \mapsto \begin{cases} K(j) \text{ if } K(j) \text{ is defined} \\ \top \text{ otherwise} \end{cases} \right]$
    for each $j$ in $S$ in sequence do
        compute $w'_j = F_j(\Pi_j(w))$
        check that $w'_j \sqsubseteq w_j$                                     (2)
        $w_j \leftarrow w'_j$
    done
    check that $w \sqsubseteq \phi$                                           (3)
```

Fig. 2. Checking algorithm

Lemma 4. *Let (K, S) be a certificate. When computing check(K, S), at each iteration of the loop, $F_j(\Pi_j(w)) \sqsubseteq w_j$ holds for every j that has already been visited once.*

Proof. We show that this property is an inductive invariant. Let $w^k \in D^n$ be the content of the variable w after the k-th iteration of the loop.

- The property is obvious at the beginning, since no j has been visited.
- Assume the invariant just before the k-th iteration. We need to prove that it holds just after. Let $j \in [1, n]$ such that j has been visited during iterations $[1, k]$. First we remark that $w^k \sqsubseteq w^{k-1}$, because of the test $w'_j \sqsubseteq w_j$ in line (2). Thus, as F_j is monotone we have $F_j(\Pi_j(w^k)) \sqsubseteq F_j(\Pi_j(w^{k-1}))$ and to show $F_j(\Pi_j(w^k)) \sqsubseteq w^k_j$ it is enough to prove $F_j(\Pi_j(w^{k-1})) \sqsubseteq w^k_j$. We consider two cases.
    - If j was visited during the k-th iteration then the assignment in the loop implies that $w^k_j = F_j(\Pi_j(w^{k-1}))$ and we conclude.
    - Otherwise j had been visited before. The invariant before iteration k thus implies that $F_j(\Pi_j(w^{k-1})) \sqsubseteq w^{k-1}_j$ and we also know that $w^k_j = w^{k-1}_j$ because j was not visited at this iteration. □

This suggests the following optimization: the test $w'_j \sqsubseteq w_j$ in the loop only needs to be done for the first occurence of j in S.

The following theorem establishes the correctness of the algorithm.

Theorem 5. *Let (K, S) be a certificate. If check(K, S) succeeds then the program satisfies the associated security property ϕ.*

Proof. We prove that when exiting from the loop, $F_j(\Pi_j(w)) \sqsubseteq w_j$ holds for every $j \in [1, n]$.

- If j appears in S, Lemma 4 applies.
- Otherwise, the line (1) of the algorithm ensures that j is not defined in K. Therefore, the initial value of w_j was $\top$. As w_j was never updated, the constraint is trivially satisfied.

This proves that the tuple w obtained at the end of the reconstruction is a post-fixpoint of F. Line (3) ensures that this is also a direct witness for ϕ. □

This verification scheme has the following benefits, compared to the naive solution of sending/verifying the whole witness:

- Abstract states need to be sent only for a subset of the program points.
- Some program points may not need to be evaluated, if they are not necessary to prove the property.
- Comparisons between abstract states are only needed for the program points for which an abstract state is sent.

5.2 Certificate generation

For a witness w, we are looking for a good certificate for the verification algorithm described above. The simplest one is (w, S) where S can be any strategy that evaluates every program point once (in any order). But, if for example F is a forward analysis and S follows the control flow graph, then most of the w_j will be overwritten before being used and therefore can be omitted. Keeping only the loop headers allows for much smaller certificates, with simple strategies. This is the core idea of the compression technique proposed in [BJP06].

We slightly generalize this setting in two ways: First, the control flow graph is more than we really need: what is required is the dependencies between the w_j which may form a sparser graph. We opt for an intermediate solution: the "static" dependency that are induced by the projections Π_j, restricted to the program points for which w has a non-$\top$ value. Second, rather than anottating loop headers, what we really want to do is to break every cycle of this dependency graph with at least one program point for which K is defined, which for some loop nestings requires stricly less of them. While it does not exploit all the generality of the checker, this strategy is optimal for generating certificates that evaluate every w_j at most once.

Definition 9. *The dependency graph of w is the directed graph $DP_w = (J_w, \rightarrow)$ whose set of vertices is $J_w = \{j \in [1, n] \mid w_j \neq \top\}$ and such that $i \rightarrow j$ iff "$i \in \Pi_j(w)$", formally $\Pi_j(x_1, \ldots, x_n) = (x_{i_{j,1}}, \ldots, x_{i_{j,k_j}})$ with $i_{j,l} = i$ for some l.* The following theorem (whose proof is omitted) formalizes the intuition that it is sufficient to break the cycles in DP_w to obtain a certificate.

Theorem 6. *Let (K, S) be a certificate such that*

- $\forall j \in \mathit{Dom}(K) \quad K(j) = w_j$ *and*

- $\forall i, j \quad i, j \in S \wedge j \to i \implies$ *j first appears before i in S* $\vee\ K(j) = w_j$ *and*
- $\forall j \quad K(j)$ *is defined* $\implies j \in S$.

Then $check(K, S)$ *succeeds.*

Therefore, generating a smallest certificate for w amounts to finding a minimal subset K of $[1, n]$ that "breaks the cycles". This is known as the feedback node set problem. While it is NP-complete in the general case, some polynomial algorithms [LL88,Koe05] exists for the particular case of *reducible* graphs, which is the case of structured control flow graphs. They run in $O(m \log(n))$ where m is the number of edges and n the number of vertices. Note that this applies to weighted graphs as well, so that it would be possible to take into account the concrete coding size needed by each program point for a particular witness.

The graph obtained from DP_w by removing the exiting arcs of every vertex in some feedback node set K naturally forms a partial order, and it is easy to see that every total order S on J_w satisfying this order meets the necessary conditions for Theorem 6 to apply, thus implying the validity of the certificate (K, S). We haven't explored the possible representation of the order S. A possible solution is to let the code consumer deduce such an order from the K part, which is trivial as soon as the user has sufficient ressources to build the reverse dependency graph.

Applying this principle to the bubble sort example of the previous section, we take the *loop* state that split the whole graph, ending with the certificate $(\{loop \to j \geq 0\}, [loop, swap])$, compared to the initial polyhedron and pruned witness that are shown in Table 1.

Finally we can justify the choice for the definition of witnesses that we tried to optimize in the previous sections. As we are sending parts of this abstract property, it is best if its size is already minimized, hence the weakened condition $F(w) \sqsubseteq \phi$ rather than $w \sqsubseteq \phi$ in the (relaxed) definition of witnesses. But we remark that the condition can be further generalized in, say $F^k(w) \sqsubseteq \phi, k \geq 0$, the limit being given by our certificate generation and verification algorithms: as the checker permits to iterate more than once, valid certificates could be obtained with the condition $F^k(w) \sqsubseteq \phi, k \geq 0$ for witnesses. However, letting $k = 1$ ensures a very simple certificate generation (Theorem 6). Note that the other constraint that w must be a post-fixpoint is crucial for the verification to succeed and cannot be weakened.

6 Related work

Albert *et al.* [AAPH06] describe a technique for reducing fixpoints produced by a generic fixpoint algorithm. The fixpoint algorithm is presented in the setting of logic program analysis but the underlying algorithmics of queues and dependence graph is common to workset-based analyses. The reduction technique monitors the fixpoint iteration to detect which program points improves other program points. The reduced certificate then consists of the fixpoint value at these program points plus data to start the fixpoint iteration. The checker takes as argument a reduced abstract property and an iteration strategy for the fixpoint algorithm and use the generic algorithm for generating the full fix-point. Thus, the certificates have the same structure as ours. The main difference is that their certificates are obtained by observing the behavior of an iterative fixpoint solving while our algorithm works by using the dependencies in the post-fixpoint once it has been produced. This means that our algorithm also allows us to compress a witness that is already much smaller than the least fixpoint whereas their approach only allows to compress the least fixpoint.

Rose [Ros03] proposes a fixpoint reconstruction algorithm for lightweight data flow graphs. The Java byte code verifier of the KVM is using this approach to check sparse certificates. The lightweight bytecode verifier is an instance of our algorithm for which the S part of the certificate specifies that the program points have to be processed in increasing order. This specialisation has the disadvantage that the number of program point in the K part of the certificate might be larger than needed. Also, the least fixpoint (i.e., the stronger one) is rebuilt, while there could be a much smaller witness that ensure the same property.

Besson, Jensen and Pichardie [BJP06] show how to certify checkers for abstract interpretation-based analyses. They propose a fixpoint reconstruction algorithm using the notion of *direct* witnesses *i.e.,* post-fixpoints that verify the property. Because our current algorithm is based on a more relaxed definition of witnesses (Definition 2), our certificates can be sparser. Moreover, Besson *et al.,* do not investigate how to optimise witnesses.

7 Conclusions

We have developed a general theory showning how invariants, issued as post-fixpoints of abstract interpretations, can be compressed to provide witnesses of particular program properties, as required *e.g.*, in proof-carrying code. In the case of distributive analyses, we have shown how an optimal (smallest) witness can be computed. For the non-distributive case (notably convex polyhedra analysis) we have shown how to compute a good approximation of minimal witnesses.

It is important to note that we are essentialy changing (pruning) the proof that we send to the code consumer, while the other compression mechanisms proposed so far keep all the informations produced by the original analysis.

The witnesses can be further compressed by only sending enough information to enable their reconstruction and hence verify their existence, as in [BJP06]. It would be interesting to apply lower level compression techniques to this setting, for example, sending only enough bits of information to resolve the “choices” that a checker has to make when rebuilding a witness, in the spirit of [NR01].

The pruning technique has been tested on invariants issued by a convex polyhedra analysis for proving simple security properties, namely the safety of array accesses in small programs and the absence of colisions in a system for controling trains. Even for those simple case studies, there is an improvement in the size of certificates.

References

[AAPH06] E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo. Reduced certificates for abstraction-carrying code. In *Proc. of the 22nd Int. Conf. on Logic Programming*, pages 163–178. Springer LNCS vol. 4079, 2006.

[BJP06] F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 364:273–291, 2006.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th ACM Symp. on Principles of programming languages*, pages 84–96. ACM Press, 1978.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In *Proc. of 5th Int. Conf. on Computer Aided Verification*, volume 697 of *LNCS*, pages 333–346. Springer-Verlag, 1993.

[Hal98] N. Halbwachs. About synchronous programming and abstract interpretation. *Science of Computer Programming*, 31(1):75–89, May 1998.

[Koe05] H. Koehler. A contraction algorithm for finding minimal feedback sets. In *Proc. of the 28th Australasian Conf. on Computer Science*, pages 165–173. Australian Computer Society, Inc., 2005.

[LL88] H. Levy and D. W. Low. A contraction algorithm for finding small cycle cutsets. *J. Algorithms*, 9(4):470–493, 1988.

[MJ81] S.S. Muchnick and N.D. Jones. *Program Flow Analysis: Theory and Application.* Prentice Hall Professional Technical Reference, 1981.

[MR90] T. Marlowe and B. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[Nec97] G. Necula. Proof-carrying code. In *Proc. of the 24th ACM Symp. on Principles of programming languages*, pages 106–119. ACM Press, 1997.

[NL98] G. Necula and P. Lee. Efficient representation and validation of proofs. In *Proc. of the 13th IEEE Symp. on Logic in Computer Science*, pages 93–104. IEEE Computer Society, 1998.

[NR01] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proc. of the 28th ACM Symp. on Principles of programming languages*, pages 142–154. ACM Press, 2001.

[Ros03] E. Rose. Lightweight bytecode verification. *J. Automated Reasoning*, 31(3-4):303–334, 2003.

[SSM04] S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In *Proc. of the 11th Static Analysis Symposium*, volume 3148 of *LNCS*, pages 53 – 68. Springer-Verlag, 2004.

[WAS03] D. Wu, A. W. Appel, and A. Stump. Foundational proof checkers with small witnesses. In *Proc. of the 5th ACM Int. Conf. on Principles and Practice of Declarative Programming*, pages 264–274. ACM Press, 2003.