Hiding Software Watermarks in Loop Structures

Conference Paper · July 2008

DOI: 10.1007/978-3-540-69166-2_12 · Source: DBLP

3 authors, including:

Mila Dalla Preda
University of Bologna
31 PUBLICATIONS **666** CITATIONS

SEE PROFILE

Roberto Giacobazzi
University of Verona
113 PUBLICATIONS **1,991** CITATIONS

SEE PROFILE

Hiding Software Watermarks in Loop Structures

Mila Dalla Preda, Roberto Giacobazzi, and Enrico Visentini

Dipartimento di Informatica, Università di Verona
Strada Le Grazie, 15 – 37134 Verona (Italy)
{mila.dallapreda,roberto.giacobazzi,enrico.visentini}@univr.it

Abstract. In this paper we propose a software watermarking technique based on the fact that different semantic instances might be abstracted in the same syntactic object. Our idea is to hide the watermark in a particular semantic instance and to distribute the corresponding syntactic construct. The extraction process uses a secret key in order to recover the information loss and reconstruct the watermark. In particular, we focus on loops and we base the embedding and extraction algorithm on the semantic understanding of loop-unrolling.

1 Introduction

Nowadays *software piracy*, i.e., the illegal reuse of proprietary code, is a key concern for software developers. *Code obfuscation*, whose aim is to obstruct code decipherment, represents a preventive tool against software piracy: attackers cannot steal what they do not understand [7,8]. Once an attacker goes beyond this defense, *software watermarking* allows the owner of the violated code to prove the ownership of the pirated copies [5,6,14,15]. Software watermarking is a technique for embedding a signature, i.e., an identifier reliably representing the owner, in a program. This allows software developers to prove their ownership by extracting their signature from the pirated copies. A good watermark has to be resilient to distortive attacks and not easy to remove [6].

Most of the existing watermarking techniques target a program feature which can assume *many* configurations, but hide the watermark in just *one* of them. Consider, for example, the watermarking technique [17] that modifies the register allocation: although there are many allocations that suit the program data flow, only one is designated to be the signature and thereby used in the marked program. The same idea applies in [14], where a distinctive permutation of basic blocks is selected among the many possible ones. Both [14] and [17] are *static* techniques, because they affect only the layout of programs. Notice that a statically watermarked program exhibits only the watermark configuration and rules out all the other ones: this may help, rather than hinder, attackers, not to mention the ease of subverting layout while preserving functionality.

Dynamic watermarking techniques exploit configurations that programs assume at runtime, thus allowing many candidate configurations to coexist in the same program. For instance, the path-based technique [4] targets the runtime branching behavior of programs: a program executes different paths on different inputs, but only the special input provides the path that outlines the signature. Likewise, the threading technique [16] yields multi-thread programs in which different configurations arise from how race conditions between threads are resolved; once again, a special input provides

M. Alpuente and G. Vidal (Eds.): SAS 2008, LNCS 5079, pp. 174–188, 2008.

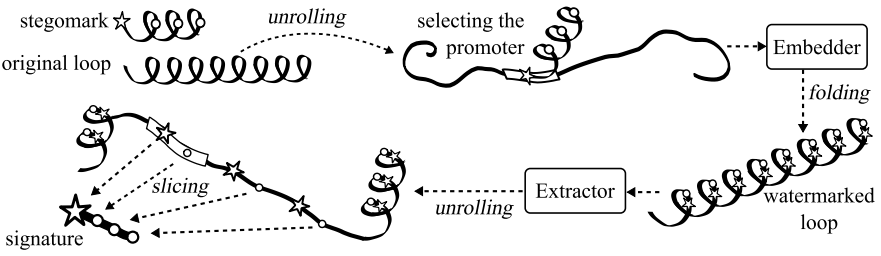

Fig. 1. Watermarking loops with loop-based watermarks

the configuration associated to the signature. Such dynamic techniques are not trivial to thwart: both branching and threading behaviors are tied to functionality, hence their distortion may result in a distortion of functionality. The coexistence of watermarked and unwatermarked configurations within the same program also characterizes the *abstract* watermarking technique [13]. Here a configuration is a parametric abstract domain saying whether a watermark variable w, which is assigned twice and computed through the Horner scheme, is constant or not. Observe that the main point is not the use of the Horner scheme but the fact that w is constant only in the domain parametrized by a key, while other domains consider w to have stochastic behavior [13].

The idea. Contrary to [13], loops are the basic block of the dynamic watermarking technique we propose in this paper. A loop is a programming construct in which a piece of code, called the loop *body*, is executed repeatedly, thus giving rise to sequences of *iterations*. In the proposed technique, *any* subsequence of such sequences is a candidate watermarking configuration. The aim is to embed, in *one* of the subsequences, a *loop-based watermark*, i.e., a watermark that is itself computed iteratively. This is done by enriching the loop body with additional code that yields the signature only within the watermarking subsequence – otherwise it does not produce significant results. Consider for example the program $s := 0$; **for** $i := n$ **to** 50 **do** $s := s + i$ **od**, which performs 50 iterations if $n = 1$. Let the *Beast Software Corporation* have signature 666, computed in 2 iterations by $W := 53$; **for** $i := 17$ **to** 18 **do** $W := W + i^2$ **od**. To watermark the former program, *Beast* moves both $W := 53$ and $W := W + i^2$ in the body of the original loop, thus obtaining program $s := 0$; **for** $i := n$ **to** 50 **do** P_i **od**, in which $P_i \triangleq [W := s-83;\ s := s+i;\ W := W+i^2]$. Expression $s-83$ evaluates precisely to 53 only when $n = 1$ and $i = 17$: these are the key values for detecting the watermarking subsequence, which spans two iterations out of 50 (those at $i = 17$ and $i = 18$). At extraction time such a subsequence is made syntactically independent from the native loop: $s := 0$; **for** $i := n$ **to** 16 **do** P_i **od**; P_i; P_{i+1}; **for** $i := 19$ **to** 100 **do** P_i **od**. What is useless for the computation of the signature is then sliced away [19]: $s := 0$; **for** $i := n$ **to** 16 **do** $s := s+i$ **od**; $W := s-83$; $W := W+i^2$; $W := W+(i+1)^2$. Here, when $n = 1$, W outputs 666. The tool we have used to make the subsequence crop out is *loop-unrolling* [2], a loop transformation that writes out iterations into sequential code, thereby making loop behavior at each iteration syntactically analyzable. As we show in Fig. 1, loop-unrolling is the core of both the embedding and extraction algorithms.

In a native loop L performing N iterations on input I, we can embed a loop-based watermark W requiring $N_W \leq N$ iterations of a code fragment M_W, called *stegomark*. By design, M_W has to get the correct initialization only when it is evaluated in a specific native iteration Δ, called *promoter*. We designate Δ by unrolling L entirely. We establish the dependence that binds M_W to Δ through program slicing [19]. Then we fold L and we insert M_W in its body, thus obtaining L_W. For an attacker now unrolling L_W is not of help in determining Δ anymore, because M_W appears in every iteration. Moreover, if L_W is contained in program P_W, any loop L' in P_W that includes a fragment of code M' matching the structure of M_W may potentially carry a watermark as well (although $L' \neq L_W$ is highly unlikely to yield a reliable signature). Thus, to retrieve the signature, for each L' we have to: (i) perform a partial unrolling which exposes, if possible, only the subsequence of N_W iterations starting from Δ; (ii) slice P_W using as criterion the code of M' included in the last iteration of the subsequence; (iii) run the slice on input I and collect the result in the set S of candidate signatures. Finally we have only to identify the signature among the elements of S. Observe that the proposed scheme allows the embedding of *any* kind of loop-based watermarks. In the specific watermarking technique we describe in Sec. 5, the iterative construction of the signature is provided by the evaluation of a polynomial through the Horner scheme as in [13]. We specify programs and their semantics following the syntax and semantics of the simple imperative language described in [12]. Syntactic program transformations, like loop-unrolling and code insertion, are related to their semantic counterpart following the abstract interpretation-based framework of Cousot and Cousot [12].

2 Preliminaries

Notation. Let $\wp(X)$ denote the *powerset* of a set X, namely the set of all subsets of X: $\wp(X) \triangleq \{Y \mid Y \subseteq X\}$. A *poset* is a set X endowed with a partial ordering $\leq_X$, denoted $\langle X, \leq_X \rangle$. Let $\bot_X$ denote, when it exists, the *minimum* of poset X, i.e., $\forall x \in X.\ \bot_X \leq_X x$. An element a is an *upper bound* of X if $\forall x \in X.\ x \leq_X a$. The minimum of the set of upper bounds of X, when it exists, is called the *least upper bound (lub)* of X and it is denoted as $\bigvee X$. A function $f : X \to Y$ from poset X to poset Y is *surjective* when $\forall y \in Y.\ \exists x \in X.\ f(x) = y$. It is $\bot_X$*-strict* when $f(\bot_X) = \bot_Y$. It is *monotonic* if $\forall x, x' \in X.\ x \leq_X x' \implies f(x) \leq_Y f(x')$. It is *additive* if it preserves the lub of every $S \subseteq X$, i.e., $f(\bigvee_X S) = \bigvee_Y f(S)$, where $f(S) \triangleq \{f(x) \mid x \in S\}$. Let $f : X \to X$ be an additive function. A *fixpoint* of f is an element $x \in X$ such that $f(x) = x$. The *least fixpoint* $\mathsf{lfp}^{\leq_X} f$ is the minimum among the fixpoints of f in X.

Abstract Interpretation. In abstract interpretation, any description of program behavior is obtained as an approximation (abstraction) of the most detailed (concrete) program specification available, which is usually a formal semantics [10,11]. Both concrete semantics and abstract behavior are computed on posets: hence there are a *concrete poset* $\langle C, \leq_C \rangle$ and an *abstract poset* $\langle A, \leq_A \rangle$, whose orderings qualitatively model relative precision between elements. When an *abstraction map* $\alpha : C \to A$ and a *concretization map* $\gamma : A \to C$ interrelate the two domains by forming an adjunction, i.e., $\forall c \in C, a \in A.\ \alpha(c) \leq_A a \iff c \leq_C \gamma(a)$, we have a *Galois connection*, denoted $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$. In particular, if α is surjective, we have a *Galois insertion*, denoted

Program Syntax

Integers $\mathtt{n} \in \mathbb{Z}$
Variables $\mathtt{Y} \in \mathbb{X}$
Arith. Exps $\mathtt{E} \in \mathbb{E}$,
$\mathtt{E} ::= \mathtt{n} \mid \mathtt{Y} \mid \mathtt{E}_1 @ \mathtt{E}_2$
Bool. Exps $\mathtt{B} \in \mathbb{B}$,
$\mathtt{B} ::= \mathtt{E}_1 \gtrless \mathtt{E}_2 \mid \mathtt{B}_1 @ \mathtt{B}_2 \mid \neg \mathtt{B} \mid \mathtt{tt} \mid \mathtt{ff}$
Actions $\mathtt{A} \in \mathbb{A}$,
$\mathtt{A} ::= \mathtt{B} \mid \mathtt{Y} := \mathtt{E} \mid \mathtt{Y} := ?$
Symbols $\mathtt{s} \in \mathbb{S}$
Labels $\mathtt{L} \in \mathbb{L} \triangleq \mathbb{N} \times \mathbb{N} \times \mathbb{S}$
Commands $\mathtt{C} \in \mathbb{C}$,
$\mathtt{C} ::= \mathtt{L} : \mathtt{A} \to \mathtt{L}'\,;$
Programs $\mathtt{P} \in \mathbb{P} \triangleq \wp(\mathbb{C})$

Program Semantics

$\mathsf{A}(\mathtt{n})\rho \triangleq n$
$\mathsf{A}(\mathtt{Y})\rho \triangleq \rho(\mathtt{Y})$
$\mathsf{A}(\mathtt{E}_1 @ \mathtt{E}_2)\rho \triangleq \mathsf{A}(\mathtt{E}_1)\rho \mathbin{@} \mathsf{A}(\mathtt{E}_2)\rho$

$\mathsf{B}(\mathtt{tt})\rho \triangleq \mathsf{tt}$
$\mathsf{B}(\mathtt{ff})\rho \triangleq \mathsf{ff}$
$\mathsf{B}(\neg \mathtt{B})\rho \triangleq \neg \mathsf{B}(\mathtt{B})\rho$
$\mathsf{B}(\mathtt{E}_1 \gtrless \mathtt{E}_2)\rho \triangleq \mathsf{A}(\mathtt{E}_1)\rho \gtrless \mathsf{A}(\mathtt{E}_2)\rho$
$\mathsf{B}(\mathtt{B}_1 @ \mathtt{B}_2)\rho \triangleq \mathsf{B}(\mathtt{B}_1)\rho \mathbin{@} \mathsf{B}(\mathtt{B}_2)\rho$

$\mathsf{S}(\mathtt{B})\rho \triangleq \{\rho' \mid \mathsf{B}(\mathtt{B})\rho' = \mathsf{tt} \wedge \rho' = \rho\}$
$\mathsf{S}(\mathtt{Y} := \mathtt{E})\rho \triangleq \{\rho[\mathtt{Y} := \mathsf{A}(\mathtt{E})\rho]\}$
$\mathsf{S}(\mathtt{Y} := ?)\rho \triangleq \{\rho' \mid \exists z \in \mathbb{Z}.\ \rho' = \rho[\mathtt{Y} := z]\}$

Program Abstractions

$\mathsf{act}(\mathtt{L} : \mathtt{A} \to \mathtt{L}'\,;) \triangleq \mathtt{A}$
$\mathsf{lab}(\mathtt{L} : \mathtt{A} \to \mathtt{L}'\,;) \triangleq \mathtt{L}$
$\mathsf{lab}(\mathtt{P}) \triangleq \bigcup_{\mathtt{C} \in \mathtt{P}} \{\mathsf{lab}(\mathtt{C})\}$
$\mathsf{suc}(\mathtt{L} : \mathtt{A} \to \mathtt{L}'\,;) \triangleq \mathtt{L}'$
$\mathsf{suc}(\mathtt{P}) \triangleq \bigcup_{\mathtt{C} \in \mathtt{P}} \{\mathsf{suc}(\mathtt{C})\}$
$\mathsf{var}(\mathtt{E}) \triangleq \{\mathtt{Y} \in \mathbb{X} \mid \mathtt{Y} \text{ is in } \mathtt{E}\}$
$\mathsf{var}(\mathtt{B}) \triangleq \{\mathtt{Y} \in \mathbb{X} \mid \mathtt{Y} \text{ is in } \mathtt{B}\}$
$\mathsf{var}(\mathtt{Y} := \mathtt{E}) \triangleq \{\mathtt{Y}\} \cup \mathsf{var}(\mathtt{E})$
$\mathsf{var}(\mathtt{Y} := ?) \triangleq \{\mathtt{Y}\}$
$\mathsf{var}(\mathtt{C}) \triangleq \mathsf{var}(\mathsf{act}(\mathtt{C}))$
$\mathsf{var}(\mathtt{P}) \triangleq \bigcup_{\mathtt{C} \in \mathtt{P}} \mathsf{var}(\mathtt{C})$

Fig. 2. Syntactic and semantic program constructs

$C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$; it can be proved that we always have a Galois insertion whenever α, γ are monotonic, $c \leq_C \gamma(\alpha(c))$ and $\alpha(\gamma(a)) = a$. Given a Galois connection $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$, a concrete function $f : C \to C$ and an abstract function $f^\sharp : A \to A$, we say that $f^\sharp$ is a *correct* approximation of f in A if $\alpha \circ f \leq_A f^\sharp \circ \alpha$. We let $f^A \triangleq \alpha \circ f \circ \gamma$ denote the *best correct approximation* of f on A. When the correctness condition is strengthened to equality, i.e., when $\alpha \circ f = f^\sharp \circ \alpha$, the abstract function $f^\sharp$ is a *complete* approximation of f on A. When α is $\bot_C$-strict and additive and $f^\sharp$ is complete wrt. f and A, then $\alpha(\mathsf{lfp}^{\leq_C} f) = \mathsf{lfp}^{\leq_A} f^\sharp$, i.e., no loss of information is accumulated in the abstract computation through $f^\sharp$ [1,9]. Then a *fixpoint transfer* can be made from C to A.

Programming Language. We consider the imperative language introduced in [12] (see Fig. 2). Any command $\mathtt{C}$ has the form $\mathtt{L} : \mathtt{A} \to \mathtt{L}'\,;$, meaning that $\mathtt{C}$ is referred to through label $\mathtt{L}$, performs action $\mathtt{A}$ and in turn refers to commands with label $\mathtt{L}'$. $\mathtt{A}$ can be either a deterministic ($\mathtt{Y} := \mathtt{E}$) or random assignment ($\mathtt{Y} := ?$), or a boolean test evaluation. A *label* or *entrypoint* $\mathtt{L} \triangleq im\mathtt{s}$ consists of an *index* $i \in \mathbb{N}$, a *memory value* $m \in \mathbb{N}$ and a symbol $\mathtt{s}$ from an alphabet $\mathbb{S}$: whenever $i, m > 0$, we have that $\mathtt{C}$ is the m-th copy of a native command $\overline{\mathtt{C}}$ at entrypoint $00\mathtt{s}$ and $\mathtt{C}$ is also member of the i-th unrolled loop (see Sect. 4). A *program* $\mathtt{P}$ is a possibly infinite set of commands [1] whose execution starts at entrypoints in $\mathfrak{L}(\mathtt{P}) \subseteq \mathsf{lab}(\mathtt{P})$. Program variables in $\mathtt{P}$ take their values in an *environment* $\rho \in \mathfrak{E}(\mathtt{P})$, which is a mapping from $\mathsf{var}(\mathtt{P}) = \mathsf{dom}\,[\rho]$ to $\mathbb{Z} \cup \{\mho\}$, where $\mho \notin \mathbb{Z}$ is the undefined value. When the domain of ρ is not relevant, we can write $\rho \in \mathfrak{E}$. As shown in Fig. 2, we use functions $\mathsf{A}(\mathtt{E}) : \mathfrak{E}(\mathtt{P}) \to \mathbb{Z} \cup \{\mho\}$ and $\mathsf{B}(\mathtt{B}) : \mathfrak{E}(\mathtt{P}) \to \{\mathsf{tt}, \mathsf{ff}, \mho\}$ to evaluate arithmetic ($\mathtt{E}$) or boolean ($\mathtt{B}$) expressions of $\mathtt{P}$; evaluation propagates $\mho$ from subexpressions to superexpressions. We also use function $\mathsf{S}(\mathtt{A}) : \mathfrak{E}(\mathtt{P}) \to \wp(\mathfrak{E}(\mathtt{P}))$, which evaluates action $\mathtt{A}$ by returning the set of environments $\mathtt{A}$ generates when executed. A *state* $s = \langle \rho, \mathtt{C} \rangle$ pairs an environment $\rho \in \mathfrak{E}(\mathtt{P})$ with a command $\mathtt{C} \in \mathtt{P}$. The set of states resulting from the execution of $\mathtt{C}$ in ρ is $\mathsf{S}(\langle \rho, \mathtt{C} \rangle) \triangleq \{\langle \rho', \mathtt{C}' \rangle \mid \mathtt{C}' \in \mathtt{P} \wedge \rho' \in \mathsf{S}(\mathsf{act}(\mathtt{C}))\rho \wedge \mathsf{suc}(\mathtt{C}) = \mathsf{lab}(\mathtt{C}')\}$; relation S models

[1] Here we follow [12] and consider programs as possibly infinite sequences of commands.

the *transition* between states. From the set $\mathfrak{L}(\mathtt{P}) \subseteq \mathsf{lab}(\mathtt{P})$ of the initial entrypoints of $\mathtt{P}$, we can define the set $\mathfrak{I}(\mathtt{P}) \triangleq \{\langle \rho, \mathtt{C}\rangle \mid \rho \in \mathfrak{E}(\mathtt{P}) \wedge \mathtt{C} \in \mathtt{P} \wedge \mathsf{lab}(\mathtt{C}) \in \mathfrak{L}(\mathtt{P})\}$ of the initial states of $\mathtt{P}$. *Trace semantics* $\mathsf{S}(\mathtt{P}) \triangleq \mathsf{lfp}^{\subseteq}\,\mathsf{F}(\mathtt{P})$ is the least fixpoint of an operator $\mathsf{F}(\mathtt{P})\mathcal{T} \triangleq \mathfrak{I}(\mathtt{P}) \cup \{\sigma s s' \mid \sigma s \in \mathcal{T} \wedge s' \in \mathsf{S}(s)\}$. Each *finite partial trace* $\sigma \in \mathsf{S}(\mathtt{P}) \subseteq \mathfrak{D}$ records a finite partial execution of $\mathtt{P}$. We let $\mathfrak{D}$ be the set of the finite partial traces of all programs and σ_j be the $(j+1)$-th state of σ. A set $\mathcal{T} \in \wp(\mathfrak{D})$ of traces can be abstracted by collecting only the commands executed along the traces [12]. Thus $\mathbb{p}(\mathcal{T}) \triangleq \{\mathtt{C} \mid \exists \sigma \in \mathcal{T}.\, \exists j \in [0, |\sigma|).\, \exists \rho \in \mathfrak{E}.\, \sigma_j = \langle \rho, \mathtt{C}\rangle\}$ induces a Galois insertion $\langle \wp(\mathfrak{D}), \subseteq\rangle \underset{\mathbb{p}}{\overset{\mathsf{S}}{\leftrightarrows}} \langle \mathbb{P}/_{\equiv}, \sqsubseteq\rangle$ which interprets programs as an abstraction of their (trace) semantics. In the abstract domain of programs, $\mathtt{P}$ and $\mathtt{Q}$ collapse ($\mathtt{P} \equiv \mathtt{Q}$) iff $\mathsf{S}(\mathtt{P}) = \mathsf{S}(\mathtt{Q})$ up to semantic equivalences between actions (for instance, $\mathtt{Y} := 6$ is semantically equivalent to $\mathtt{Y} := 2 \times 3$). The *syntactic refinement* $\mathtt{P} \sqsubseteq \mathtt{Q}$ holds iff $\mathsf{S}(\mathtt{P}) \subseteq \mathsf{S}(\mathtt{Q})$ up to semantic equivalences between actions.

Principle of Program Transformation. Any syntactic program transformer $\mathbb{t}$, altering the code of $\mathtt{P}$ and returning new program $\mathtt{P}'$, induces a corresponding semantic transformer t turning $\mathsf{S}(\mathtt{P})$ into $\mathsf{S}(\mathtt{P}')$ [12]. If we let $t(\mathcal{T}) \triangleq \{t(\sigma) \mid \sigma \in \mathcal{T}\}$, then t induces a Galois connection $\langle \wp(\mathfrak{D}), \subseteq\rangle \underset{t}{\overset{\gamma_t}{\leftrightarrows}} \langle \wp(\mathfrak{D}), \subseteq\rangle$, where t acts as an abstraction. By composing the two Galois connection introduced so far, we can derive a similar notion about $\mathbb{t}$, i.e., $\langle \mathbb{P}/_{\equiv}, \sqsubseteq\rangle \underset{\mathbb{t}}{\overset{\gamma_{\mathbb{t}}}{\leftrightarrows}} \langle \mathbb{P}/_{\equiv}, \sqsubseteq\rangle$ [12]. This kind of composition allows us to derive $\mathbb{t}$ as the best correct approximation of semantic transformer t, i.e., $\mathbb{t} \sqsupseteq \mathbb{p} \circ t \circ \mathsf{S}$. In particular, when the transformation is decidable, we have $\mathbb{t} \equiv \mathbb{p} \circ t \circ \mathsf{S}$. The systematic design of $\mathbb{t}$ from t takes advantage of fixpoint transfers [12]. In the following we consider only decidable transformations, such as loop-unrolling and assignment-insertion. Hence, we derive $\mathbb{t} \triangleq \mathsf{lfp}^{\sqsubseteq}\,\mathbb{F}^t$ in fixpoint form by combining $\mathbb{t} \equiv \mathbb{p} \circ t \circ \mathsf{S}$ with the following equivalence: $\mathbb{p} \circ t \circ \mathsf{S} = \mathbb{p} \circ t \circ \mathsf{lfp}^{\subseteq}\,\mathsf{F} = \mathbb{p} \circ \mathsf{lfp}^{\subseteq}\,\mathsf{F}^t \equiv \mathsf{lfp}^{\sqsubseteq}\,\mathbb{F}^t$. Notice that the first equality follows by definition of S; the other ones hold only if operators $\mathsf{F}^t : \wp(\mathfrak{D}) \to \wp(\mathfrak{D})$ and $\mathbb{F}^t : \mathbb{P}/_{\equiv} \to \mathbb{P}/_{\equiv}$ are designed to fit the requirements of the fixpoint transfers applied within Galois connections $\langle \wp(\mathfrak{D}), \subseteq\rangle \underset{t}{\overset{\gamma_t}{\leftrightarrows}} \langle \wp(\mathfrak{D}), \subseteq\rangle$ and $\langle \wp(\mathfrak{D}), \subseteq\rangle \underset{\mathbb{p}}{\overset{\mathsf{S}}{\leftrightarrows}} \langle \mathbb{P}/_{\equiv}, \sqsubseteq\rangle$ respectively. The correctness of $\mathbb{t}$ is formalized through some *observational abstraction* $\alpha_{\mathcal{O}}$ such that $\langle \wp(\mathfrak{D}), \subseteq\rangle \underset{\alpha_{\mathcal{O}}}{\overset{\gamma_{\mathcal{O}}}{\leftrightarrows}} \langle \mathcal{D}_{\mathcal{O}}, \sqsubseteq_{\mathcal{O}}\rangle$. Transformer $\mathbb{t}$ is correct wrt. $\alpha_{\mathcal{O}}$ if and only if for every program $\mathtt{P} \in \mathbb{P}$, $\alpha_{\mathcal{O}}(\mathsf{S}(\mathtt{P})) = \alpha_{\mathcal{O}}(t(\mathsf{S}(\mathtt{P})))$ [12].

3 Assignment-Insertion

Let us define in the framework described above the transformation of assignment-insertion that we exploit for the embedding of M_W. Suppose we wish to insert, at entrypoint $\mathtt{J} \in \mathsf{lab}(\mathtt{P})$, an assignment $\mathtt{W} := \mathtt{E}$; we use $\mathtt{J}, \mathtt{J}' \ldots$ to denote labels targeted for insertion and $\mathtt{L}, \mathtt{L}' \ldots$ to denote other labels. Syntactically, the solution is straightforward (see Fig. 3): we modify in $\mathtt{P}$ every command referring to $\mathtt{J}$ so that now it refers to a new $\underline{\mathtt{J}} \notin \mathsf{lab}(\mathtt{P})$, then we insert in $\mathtt{P}$ a new command $\underline{\mathtt{C}} \triangleq \underline{\mathtt{J}}\mathtt{:}\ \mathtt{W} := \mathtt{E} \to \mathtt{J};$,

```
      // Original program P
00e   Z := 0;
00f   X := 0;
      // Native for-loop F
00g   ¬(X < I0) → end;
00g   X < I0 → 00h;
00h     Y := Fib(X + I1);
00v     Z := Z + Y × Y;
00i     X := X + 1 → 00g;
```

```
      // Watermarked program P'
   00e   Z := 0;
   00f   X := 0;
         // for-loop F'
   00g   ¬(X < I0) → end;
   00g   X < I0 → 00h;
   00h     Y := Fib(X + I1);
→  00v     W := (215 − Y) × 259;
   00v     Z := Z + Y × Y;
→  00i     W := 14 × W + 245760;
   00i     X := X + 1 → 00g;
```

Assignment-insertion turns P into P′. Then P′ yields program Q provided that its for-loop F′ is unrolled with $\mathbf{u} = \langle 1, 3\rangle$ and $\boldsymbol{\ell} = \langle 5, 2\rangle$. The entrypoint of each program is 00e. The new assignments carry the watermark of signature $\mathfrak{s} = 120736$. To extract $\mathfrak{s}$, we need to run on P′ the algorithm described in Sec. 5. The algorithm computes Q and try to detect a candidate assignment (★). The copies (⋆) of that assignment are discarded. In resulting program Q′ it determines (→ ○) backward-slicing criterion C and computes slicing S. On the key input, the commands in S yield signature $\mathfrak{s}$ embedded in P′.

```
                  // Program Q
                  // generated at extraction time
        00e       Z := 0;
   S    00f       X := 0;
                  // unrolled for-loop F1 (u1 = 1)
   S    10g       ¬(X < I0 − 5) → 20g;
   S    10g       X < I0 − 5 → 10h;
        10h         Y := Fib(X + I1);
        10v         W := (215 − Y) × 259;
        10v         Z := Z + Y × Y;
        10i         W := 14 × W + 245760;
   S    10i         X := X + 1 → 10g;
                  // unrolled for-loop F2 (u2 = 3)
   S    20g       ¬(X < I0 − 4) → 00g;
   S    20g       X < I0 − 4 → 20h;
   S    20h         Y := Fib(X + I1);
★  S    20v         W := (215 − Y) × 259;
        20v         Z := Z + Y × Y;
○  S    20i         W := 14 × W + 245760;
        20i         tt;
        21g         tt;
        21h         Y := Fib((X + 1) + I1);
⋆       21v         W := (215 − Y) × 259;
        21v         Z := Z + Y × Y;
○  S    21i         W := 14 × W + 245760;
        21i         tt;
        22g         tt;
        22h         Y := Fib((X + 2) + I1);
⋆       22v         W := (215 − Y) × 259;
        22v         Z := Z + Y × Y;
→ ○ S   22i         W := 14 × W + 245760;
   S    22i         X := X + 3 → 20g;
                  // for-loop F of program P'
        00g       [...]
```

Fig. 3. Three programs

obtaining P′. In case $\mathtt{W} \in \mathsf{var}(\mathtt{P})$, however, $\mathsf{S}(\mathtt{P}')$ may differ a lot from $\mathsf{S}(\mathtt{P})$, as the new value of W may alter deeply the evaluations of subsequent boolean conditions and control flow of P. To prevent these major changes, we might restore the value of W before W is used again. Alternatively, we just exploit a variable W that is either *fresh* wrt. P, namely $\mathtt{W} \notin \mathsf{var}(\mathtt{P})$, or *dead* wrt. entrypoint J targeted for the insertion, i.e., commands executed after the reaching of J must not define or use W any longer. Under this hypothesis, $\langle \rho_0, \mathtt{L}\colon \mathtt{A}_0 \to \mathtt{J};\rangle\langle \rho_1, \mathtt{J}\colon \mathtt{A}_1 \to \mathtt{L}';\rangle \in \mathsf{S}(\mathtt{P})$ can be transformed into $\langle \varphi_0, \mathtt{L}\colon \mathtt{A}_0 \to \underline{\mathtt{J}};\rangle\langle \varphi_1, \underline{\mathtt{C}}\rangle\langle \varphi_1', \mathtt{J}\colon \mathtt{A}_1 \to \mathtt{L}';\rangle \in \mathsf{S}(\mathtt{P}')$. We let φ_0, φ_1 be ρ_0, ρ_1 enriched with $\mathtt{W} \mapsto \mho$ in case W is fresh. On the other side, we let $\varphi_1' \triangleq \rho_1 \oplus \{\mathtt{W} \mapsto \mathsf{A}(\mathtt{E})\rho_1\}$, meaning that W has to belong to $\mathsf{dom}\,[\varphi_1']$ and has to take value $\mathsf{A}(\mathtt{E})\rho_1$. We say that φ_1' is an enhancement of ρ_1. In general, given $\rho \in \mathfrak{E}$, its enhancement $\rho \oplus \omega \triangleq (\rho \setminus \omega) \cup \omega$

$$t^{\text{in}}(\varepsilon\langle\rho,\mathtt{C}\rangle) \triangleq t^{\text{in}}(\langle\rho,\mathtt{C}\rangle, \{\mathtt{W_J} \leftarrow \mho \mid \mathtt{W_J} \in \mathcal{W} \wedge \mathtt{W_J} \notin \mathsf{dom}\,[\rho]\})$$

$$t^{\text{in}}(\sigma'\langle\rho',\mathtt{C}'\rangle\langle\rho,\mathtt{C}\rangle) \triangleq \textbf{let } \overline{\sigma}\langle\overline{\rho},\overline{\mathtt{C}}\rangle = t^{\text{in}}(\sigma'\langle\rho',\mathtt{C}'\rangle) \textbf{ in } \overline{\sigma}\langle\overline{\rho},\overline{\mathtt{C}}\rangle t^{\text{in}}(\langle\rho,\mathtt{C}\rangle, \overline{\rho} \text{ restricted to domain } \mathcal{W})$$

$$t^{\text{in}}(\langle\rho,\mathtt{C}\rangle,\omega) \triangleq \textbf{let } \varphi = \rho \oplus \omega \textbf{ in match } t^{\text{in}}(\mathtt{C}) \textbf{ with}$$

$$\mathtt{J}\colon \mathtt{W_J} := \mathtt{E_J} \to \mathtt{L}'\,;\mathtt{C}' \longmapsto \langle\varphi, \mathtt{J}\colon \mathtt{W_J} := \mathtt{E_J} \to \mathtt{L}'\,;\rangle\langle\varphi[\mathtt{W_J} := \mathsf{A}(\mathtt{E_J})\omega], \mathtt{C}'\rangle$$

$$\mathtt{C}' \longmapsto \langle\varphi, \mathtt{C}'\rangle$$

$$t^{\text{in}}(\mathtt{J}\colon \mathtt{A} \to \mathtt{J}'\,;) \triangleq \underline{\mathtt{J}}\colon \mathtt{W_J} := \mathtt{E_J} \to \mathtt{J}\,;\mathtt{J}\colon \mathtt{A} \to \underline{\mathtt{J}}'\,; \qquad t^{\text{in}}(\mathtt{L}\colon \mathtt{A} \to \mathtt{J}'\,;) \triangleq \mathtt{L}\colon \mathtt{A} \to \underline{\mathtt{J}}'\,;$$

$$t^{\text{in}}(\mathtt{J}\colon \mathtt{A} \to \mathtt{L}'\,;) \triangleq \underline{\mathtt{J}}\colon \mathtt{W_J} := \mathtt{E_J} \to \mathtt{J}\,;\mathtt{J}\colon \mathtt{A} \to \mathtt{L}'\,; \qquad t^{\text{in}}(\mathtt{L}\colon \mathtt{A} \to \mathtt{L}'\,;) \triangleq \mathtt{L}\colon \mathtt{A} \to \mathtt{L}'\,;$$

Fig. 4. Semantic assignment-insertion

augments ρ with the mappings in $\omega \in \mathfrak{E}$, overwriting $\mathsf{A}(\mathtt{W})\rho$ with $\mathsf{A}(\mathtt{W})\omega$ in case there is a clash on $\mathtt{W}$. Due to the triviality of the trace transformation, observational abstraction $\alpha^{\text{in}}_{\mathcal{O}}$ for assignment-insertion needs only to discard the inserted states and return the resulting sequence, expunged from environments:

$$\alpha^{\text{in}}_{\mathcal{O}}(\sigma) \triangleq \lambda j.\ \alpha^{\text{in}}_{\mathcal{O}}(\sigma_j) \qquad \alpha^{\text{in}}_{\mathcal{O}}(im\mathtt{s}\colon \mathtt{A} \to \mathtt{L}'\,;) \triangleq im\mathtt{s}\colon \mathtt{A} \to \mathtt{L}'\,;$$

$$\alpha^{\text{in}}_{\mathcal{O}}(\langle\rho,\mathtt{C}\rangle) \triangleq \alpha^{\text{in}}_{\mathcal{O}}(\mathtt{C}) \qquad \alpha^{\text{in}}_{\mathcal{O}}(im\underline{\mathtt{s}}\colon \mathtt{A} \to \mathtt{L}'\,;) \triangleq \varepsilon\ .$$

Semantic transformation t^{in} for assignment-insertion, shown in Fig. 4, scans the traces of $\mathtt{P}$ state by state, performing different insertions. Each time it finds entrypoint $\mathtt{J}$, it inserts correspondent assignment $\mathtt{W_J} := \mathtt{E_J}$. We have that each $\mathtt{J}$ is a target label in $\mathtt{P}$, each $\mathtt{W_J}$ is a variable from a set $\mathcal{W}$ and each $\mathtt{E_J}$ is such that $\mathsf{var}(\mathtt{E_J}) \subseteq \mathsf{var}(\mathtt{P}) \cup \mathcal{W}$. The algorithm also enhances the environments of the trace, replacing each ρ with $\rho \oplus \omega$. To this purpose, it maintains a special environment ω which changes dynamically from state to state. At the beginning, it tracks only the fresh variables in $\mathcal{W}$, mapping each one to $\mho$. In the following, after a state has been transformed, it tracks all variables in $\mathcal{W}$, deriving their values from the (enhanced) environment of the transformed state. Since each $\mathtt{W_J} \in \mathcal{W}$ is dead at entrypoint $\mathtt{J}$, the enhancement of ρ influence neither the evaluation of arithmetic and boolean expressions, nor the control flow, as desired. We can formally prove this fact by taking advantage of $\alpha^{\text{in}}_{\mathcal{O}}$.

Proposition 1. *Let* $\mathtt{P}$ *be a program and* $\sigma \in \mathsf{S}(\mathtt{P})$. *Then* $t^{\text{in}}(\sigma)$ *is a trace and* $\alpha^{\text{in}}_{\mathcal{O}}(t^{\text{in}}(\sigma)) = \alpha^{\text{in}}_{\mathcal{O}}(\sigma)$. *Furthermore* $\mathbb{p} \circ (t^{\text{in}} \circ \mathsf{F}) = \mathbb{F}^{\text{in}} \circ \mathbb{p}$.

The fixpoint transfer inside the proposition allows us to derive from t^{in} a syntactic algorithm $\mathbb{t}^{\text{in}}$ for assignment-insertion. We express it as follows:

$$\textsc{Init}^{\text{in}}(\mathtt{P}) \triangleq \{\mathtt{C}' \mid \exists \mathtt{C} \in \mathtt{P}.\ \mathsf{lab}(\mathtt{C}) \in \mathfrak{L}(\mathtt{P}) \ \wedge \mathtt{C}' \text{ is in } t^{\text{in}}(\mathtt{C})\}$$

$$\textsc{Next}^{\text{in}}(\mathtt{P})(\mathtt{Q}) \triangleq \{\mathtt{C}' \mid \exists \mathtt{C} \in \mathtt{P}.\ \exists \mathtt{D} \in \mathtt{Q}.\ \mathtt{C}' \text{ is in } t^{\text{in}}(\mathtt{C}) \wedge \mathsf{lab}(\mathtt{C}) = im\mathtt{s} \wedge (\mathsf{suc}(\mathtt{D}) = im\mathtt{s} \vee \mathsf{suc}(\mathtt{D}) = im\underline{\mathtt{s}})\}$$

$$\textsc{Iter}^{\text{in}}(\mathtt{P})(\mathtt{Q}) \triangleq \textbf{ let } \mathtt{Q}' = \mathtt{Q} \cup \textsc{Next}^{\text{in}}(\mathtt{P})(\mathtt{Q}) \textbf{ in if } \mathtt{Q}' = \mathtt{Q} \textbf{ then } \mathtt{Q}' \textbf{ else } \textsc{Iter}^{\text{in}}(\mathtt{P})(\mathtt{Q}') \textbf{ fi}.$$

4 Loop-Unrolling

The easiest looping constructs to unroll are `for`-loops. Program P of Fig. 3 includes a `for`-loop F. This loop, on input $\langle \mathcal{I}_0, \mathcal{I}_1 \rangle$, sums in Z the squares of the numbers which, in the Fibonacci sequence, have indexes from $\mathcal{I}_0$ to $\mathcal{I}_0 + \mathcal{I}_1 - 1$: if e.g. $\mathcal{I}_0 = 4$ and $\mathcal{I}_1 = 3$, then Z finally evaluates to $3^2 + 5^2 + 8^2 = 98$. Whenever a program P includes a `for`-loop F, we write $\mathtt{F} \in \mathsf{fors}(\mathtt{P})$. More formally, $\mathtt{F} \in \mathsf{fors}(\mathtt{P})$ iff $\mathtt{F} \subseteq \mathtt{P}$ and $\mathtt{F} \triangleq \{\mathtt{G}, \bar{\mathtt{G}}, \mathtt{I}\} \cup \mathtt{H}$. The couple of commands $\mathtt{G} \triangleq \mathtt{g{:}\ X} < \dot{\mathtt{E}} \rightarrow \mathtt{h;}$ and $\bar{\mathtt{G}} \triangleq \mathtt{g{:}}\ \neg(\mathtt{X} < \dot{\mathtt{E}}) \rightarrow \mathtt{p;}$, with $\mathtt{g} \neq \mathtt{h}$ and $\mathtt{g} \neq \mathtt{p}$, implements a branching named *guard*. As F always starts with the evaluation of its guard, we have $\mathfrak{L}(\mathtt{F}) = \{\mathtt{g}\}$, $\mathsf{lab}(\mathtt{F}) \cap \mathsf{lab}(\mathtt{P} \setminus \mathtt{F}) = \emptyset$ and $\mathsf{suc}(\mathtt{P} \setminus \mathtt{F}) \cap \mathsf{lab}(\mathtt{F}) \subseteq \{\mathtt{g}\}$. The guard is satisfied as long as $\mathtt{X} \in \mathbb{X}$ is less[2] than $\dot{\mathtt{E}} \in \mathbb{E}$. If the guard is not satisfied, the `for`-loop ends transferring the control flow at entrypoint $\mathtt{p} \notin \mathsf{lab}(\mathtt{F})$. Otherwise, the execution goes on through H, a set of commands named *body*, and eventually through an *increment* command $\mathtt{I} \triangleq \mathtt{i{:}\ X} := \mathtt{X} + \ddot{\mathtt{E}} \rightarrow \mathtt{g;}$, with $\mathtt{i} \neq \mathtt{g}$ and $\mathtt{i} = \mathtt{h} \vee \mathtt{i} \in \mathsf{suc}(\mathtt{H})$; notice that I makes the control flow return to the guard again. We formally define H as the collection of all the commands of P that are reachable from G without going through I, i.e., $\mathtt{H} \triangleq \mathsf{lfp}^{\subseteq}\, \mathsf{flow}(\mathtt{P})(\mathtt{Q}) \triangleq \{\mathtt{C} \in \mathtt{P} \setminus \{\mathtt{I}\} \mid \mathsf{lab}(\mathtt{C}) = \mathsf{suc}(\mathtt{G}) \vee \exists \mathtt{C}' \in \mathtt{Q}.\ \mathsf{lab}(\mathtt{C}) = \mathsf{suc}(\mathtt{C}')\}$. We require $\mathtt{g}, \mathtt{i} \notin \mathsf{lab}(\mathtt{H})$. We expect both X and the variables in $\dot{\mathtt{E}}$ and $\ddot{\mathtt{E}}$ not to be assigned inside H. We require X not to be used in $\dot{\mathtt{E}}$ or $\ddot{\mathtt{E}}$.

Finite partial trace $\langle \rho, \mathtt{G} \rangle \eta \langle \rho', \mathtt{I} \rangle \in \mathsf{S}(\mathtt{F})$ is an *iteration of* `for`*-loop* F, where $\eta \in \mathsf{S}(\mathtt{H})$; if $\mathtt{H} = \emptyset$ then $\eta = \varepsilon$. A maximal trace $\sigma \in \mathsf{S}(\mathtt{F})$ is a sequence of terminating iterations[3] followed by a state with command $\bar{\mathtt{G}}$. Along the trace, the values of $\dot{\mathtt{E}}$ and $\ddot{\mathtt{E}}$ do not change, while the value of X, though constant throughout each iteration, increases by $\ddot{\mathtt{E}}$ from one iteration to another. Thus, if ρ is an environment in a state of $\sigma \in \mathsf{S}(\mathtt{F})$, we can predict how many increments X still has to undergo, i.e., the number of the iterations from ρ till the end of σ. Let $\dot{e} \triangleq \mathsf{A}(\dot{\mathtt{E}})\rho$, $\ddot{e} \triangleq \mathsf{A}(\ddot{\mathtt{E}})\rho$ and $x \triangleq \mathsf{A}(\mathtt{X})\rho$. We just need to define $\alpha_{\mathtt{F}} : \mathfrak{C}(\mathtt{F}) \rightarrow \mathbb{N}$ such that $\alpha_{\mathtt{F}}(\rho) \triangleq \left\lfloor \frac{(\dot{e}-x)+(\ddot{e}-1)}{\ddot{e}} \right\rfloor$ if $\dot{e} \geq x$ and $\alpha_{\mathtt{F}}(\rho) \triangleq 0$ otherwise. We let ι be the total number of iterations of σ.

Along $\sigma \in \mathsf{S}(\mathtt{F})$ iterations are naturally unfolded, i.e., they come sequentially one after another. In $\mathbb{p}(\{\sigma\})$ they fold because any command $\mathtt{C} \in \mathtt{F}$, although occurring in many different iterations, always appears with the same entrypoint $\mathsf{lab}(\mathtt{C})$. In the proposed watermarking technique, folding has to be neutralized at embedding/extraction time. Loop-unrolling [2] is good at this task because it changes labels in the following way: given the so-called *unrolling factor* $u \in \mathbb{N}$, it makes all and only the occurrences of C at iterations $k \pmod u$ have the same label (with $0 \leq k < \iota$), thus partitioning the iterations of σ into u classes. Only iterations from the same class fold together. So the code of the unrolled loop is u times longer than F and each of its iterations sequentially executes the task of u native iterations. Consider for instance `for`-loop $\mathtt{F}' \in \mathsf{fors}(\mathtt{P}')$ in Fig. 3, which has a command $\overline{\mathtt{C}}$ with entrypoint 00h. Clearly $\overline{\mathtt{C}}$ appears in every iteration of any $\sigma \in \mathsf{S}(\mathtt{F}')$. Now let $\sigma = \sigma'\sigma''$, where σ' encompasses the first $0 \leq \iota_1 \leq \iota$ iterations and σ'' the last $\iota_2 = \iota - \iota_1$ ones. To unroll σ'' with factor $u_2 = 3$, we scan

[2] For short, we ignore similar kinds of `for`-loops, which use $>$, $\leq$ or $\geq$ as comparison operator.

[3] An iteration also might not conclude: this occurs when the execution of F gets trapped inside some non-terminating loops possibly included in H. In such a case none of the partial traces of $\mathsf{S}(\mathtt{F})$ can be recognized as a maximal trace which fully outlines the entire execution.

$$\begin{array}{rcl}
\mathtt{Y} := ?[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] & \triangleq & \mathtt{Y} := ? \\
\mathtt{Y} := \mathtt{E}[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] & \triangleq & \mathtt{Y} := \mathtt{E}[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] \\
\mathtt{B}_1 \mathbin{@} \mathtt{B}_2[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] & \triangleq & \mathtt{B}_1[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] \mathbin{@} \mathtt{B}_2[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] \\
\neg\mathtt{B}[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] & \triangleq & \neg\mathtt{B}[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] \\
{}^{\mathtt{tt}}/_{\mathtt{ff}}[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] & \triangleq & {}^{\mathtt{tt}}/_{\mathtt{ff}} \\
\mathtt{E}_1 \mathbin{@} \mathtt{E}_2[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] & \triangleq & \mathtt{E}_1[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] \mathbin{@} \mathtt{E}_2[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] \\
\mathtt{n}[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] & \triangleq & \mathtt{n} \\
\mathtt{Y}[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] & \triangleq & \mathtt{Y} \quad (\mathtt{Y} \neq \mathtt{X}) \\
\mathtt{X}[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] & \triangleq & \mathtt{X} + m \times \ddot{\mathtt{E}} \quad (m \neq 0) \\
\mathtt{X}[(\mathtt{X}+0\times\ddot{\mathtt{E}})/\mathtt{X}] & \triangleq & \mathtt{X}
\end{array}$$

$$\mathsf{I}(i,\mathtt{C}) \triangleq \begin{cases} i+1 & \text{if } i < I \wedge (\mathtt{C} = \bar{\mathtt{G}} \\ & \quad \vee (\mathtt{C} \notin \mathtt{F} \wedge \mathsf{suc}(\mathtt{C}) = 00\mathtt{g})) \\ 0 & \text{if } i = I \wedge \mathtt{C} = \bar{\mathtt{G}} \\ i & \text{otherwise} \end{cases}$$

$$\mathsf{M}(m,i,\mathtt{I}) \triangleq \begin{cases} m+1 & \text{if } m \in [0, u_i - 1) \\ 0 & \text{if } m = u_i - 1 \end{cases}$$

$$\mathsf{M}(m,i,\mathtt{C}) \triangleq m \qquad \text{if } \mathtt{C} \neq \mathtt{I}$$

$$t^{\mathsf{lu}}(\varepsilon\langle\rho,\mathtt{C}\rangle) \triangleq \textbf{let } i = \textbf{if } \mathtt{C} \in \mathtt{F} \textbf{ then } 1 \textbf{ else } 0 \textbf{ in } t^{\mathsf{lu}}(\langle\rho,\mathtt{C}\rangle, 0, i) \textbf{ fi}$$

$$t^{\mathsf{lu}}(\sigma'\langle\rho',\mathtt{C}'\rangle\langle\rho,\mathtt{C}\rangle) \triangleq \textbf{let } \overline{\sigma}\langle\overline{\rho},\mathtt{L}\colon \mathtt{A} \to ims\,;\rangle = t^{\mathsf{lu}}(\sigma'\langle\rho',\mathtt{C}'\rangle) \textbf{ in } \overline{\sigma}\langle\overline{\rho},\mathtt{L}\colon \mathtt{A} \to ims\,;\rangle t^{\mathsf{lu}}(\langle\rho,\mathtt{C}\rangle, m, i)$$

$$t^{\mathsf{lu}}(\langle\rho,\mathtt{C}\rangle, m, i) \triangleq \mathsf{V}(\rho[\mathtt{X} := \mathsf{A}(\mathtt{X})\rho - m\mathsf{A}(\ddot{\mathtt{E}})\rho], t^{\mathsf{lu}}(\mathtt{C}, m, i))$$

$$t^{\mathsf{lu}}(00\mathtt{s}\colon \mathtt{A} \to 00\mathtt{s}'\,;, m, i) \triangleq \textbf{let } \langle m', i'\rangle = \langle \mathsf{M}(m,i,00\mathtt{s}\colon \mathtt{A} \to 00\mathtt{s}'\,;), \mathsf{I}(i,00\mathtt{s}\colon \mathtt{A} \to 00\mathtt{s}'\,;)\rangle \textbf{ in}$$
$$\textbf{let } \langle \mathtt{L}, \mathtt{L}'\rangle = \langle ims, i'm'\mathtt{s}'\rangle \textbf{ in let } \mathtt{B} = \mathtt{X} < \dot{\mathtt{E}} - (\ell_i + u_i - 1) \times \ddot{\mathtt{E}} \textbf{ in match } \mathtt{C} \textbf{ with}$$
$$\begin{array}{rcl}
\bar{\mathtt{G}} & \longmapsto & \textbf{if } m > 0 \textbf{ then } \mathtt{L}\colon \mathtt{ff} \to \mathtt{L}\,; \textbf{ else if } i = 0 \textbf{ then } \mathtt{L}\colon \neg\mathtt{B} \to \mathtt{L}'\,; \\
 & & \quad \textbf{else } \mathtt{L}\colon \neg\mathtt{B} \to i'm'\mathtt{g}\,;\ t^{\mathsf{lu}}(\bar{\mathtt{G}}, m', i') \textbf{ fi} \\
\mathtt{G} & \longmapsto & \textbf{if } m > 0 \textbf{ then } \mathtt{L}\colon \mathtt{tt} \to \mathtt{L}'\,; \textbf{ else if } i = 0 \textbf{ then } \mathtt{L}\colon \mathtt{B} \to \mathtt{L}'\,; \\
 & & \quad \textbf{else let } i'' = \mathsf{I}(i,\bar{\mathtt{G}}) \textbf{ in } \mathtt{L}\colon \mathtt{B} \to \mathtt{L}'\,;\ \mathtt{L}\colon \neg\mathtt{B} \to i''m'\mathtt{g}\,;\ t^{\mathsf{lu}}(\mathtt{G}, m', i'') \textbf{ fi} \\
\mathtt{I} & \longmapsto & \textbf{if } m = u_i - 1 \textbf{ then } \mathtt{L}\colon \mathsf{act}(\mathtt{C})[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] \to \mathtt{L}'\,; \textbf{ else } \mathtt{L}\colon \mathtt{tt} \to \mathtt{L}'\,; \textbf{ fi} \\
_ & \longmapsto & \mathtt{L}\colon \mathsf{act}(\mathtt{C})[(\mathtt{X}+m\times\ddot{\mathtt{E}})/\mathtt{X}] \to \mathtt{L}'\,;
\end{array}$$

$$\mathsf{V}(\rho, \mathtt{ls}) \triangleq \textbf{match } \mathtt{ls} \textbf{ with}$$
$$\mathtt{L}\colon \mathtt{B} \to \mathtt{L}'\,;\ \mathtt{L}'\colon \neg\mathtt{B} \to \mathtt{L}''\,;\ \mathtt{ls}' \longmapsto \textbf{ if } \mathsf{B}(\mathtt{B})\rho \textbf{ then } \langle\rho, \mathtt{L}\colon \mathtt{B} \to \mathtt{L}'\,;\rangle \textbf{ else } \langle\rho, \mathtt{L}'\colon \neg\mathtt{B} \to \mathtt{L}''\,;\rangle\mathsf{V}(\rho, \mathtt{ls}') \textbf{ fi}$$
$$\mathtt{C}\ \mathtt{ls}' \longmapsto \langle\rho,\mathtt{C}\rangle\mathsf{V}(\rho, \mathtt{ls}')$$
$$\varepsilon \longmapsto \varepsilon$$

Fig. 5. Semantic loop-unrolling

the iterations of σ'' by triplets; for each triplet, we set the memory value of $\mathsf{lab}(\overline{\mathtt{C}})$ to 0 in the first iteration, 1 in the second iteration and to 2 in the third iteration. If we fold the new trace, we obtain `for`-loop $\mathtt{F}_2$ of program $\mathtt{Q}$ in Fig. 3: here three copies of $\overline{\mathtt{C}}$ coexists at entrypoints 20h, 21h and 22h. Similarly, by unrolling σ' with the trivial factor $u_1 = 1$, we get `for`-loop $\mathtt{F}_1$ of $\mathtt{Q}$, in which the only one copy of $\overline{\mathtt{C}}$ is located at 10h. All the copies of $\overline{\mathtt{C}}$ have the same symbol h. As index value, they use a number identifying the unrolled loop which they are member of. The fact that the code of the unrolled loops actually implements tuples of native iterations is essential to the proposed watermarking technique. We hide signatures in iterations, which are semantic objects. However, the embedder and the extractor are automatic tools that cannot deal with semantics. But they can deal with code. Thus if we define loop-unrolling as a semantic transformation and then we abstract it to a syntactic transformation [12], we can

safely rely on loop-unrolling to both embed and extract signatures. In our last example we unrolled $\sigma = \sigma'\sigma''$ using u_1 only for σ'. To attain this, we kept on unrolling σ only while $\mathtt{X} < \dot{\mathtt{E}} - (\ell_1 + u_1 - 1) \times \ddot{\mathtt{E}}$ was true, where we let $\ell_1 = \iota_2$. This approach was supported by the following proposition. Define $\ell \in [0, \iota]$ to be the *lessening factor*. Let $g(u, \ell) \triangleq \ell + u - 1$ and $\hat{\mathsf{B}} \triangleq \mathtt{X} < \dot{\mathtt{E}} - g(u, \ell) \times \ddot{\mathtt{E}}$. Let $\rho \in \mathfrak{E}(\mathtt{F})$ be an environment in σ.

Proposition 2. $\mathsf{B}(\hat{\mathsf{B}})\rho = \mathsf{ff}$ *if and only if* $0 \leq \alpha_{\mathtt{F}}(\rho) \leq g(u, \ell)$*, i.e.,* $\hat{\mathsf{B}}$ *gets false in* σ *at the last but* $g(u, \ell)$ *iteration. Moreover* $\lfloor {}^{(\iota-\ell)}/_{u} \rfloor\, u - u < \iota - g(u, \ell) \leq \lfloor {}^{(\iota-\ell)}/_{u} \rfloor\, u$.

So the unrolling of σ with u_1, ℓ_1 involved just the first $\lfloor {}^{(\iota-\ell_1)}/_{u_1} \rfloor\, u_1$ iterations. This did not keep us from unrolling unprocessed iterations with new factors $u_2 = 3$ and $\ell_2 = 0$.

As we know, loop-unrolling affects labels. Consider again `for`-loop $\mathtt{F}_2$ in Fig. 3: in iterations $k \pmod{u_2}$ each $00\mathtt{s}$ was replaced with $2m\mathtt{s}$, where $m \triangleq k \bmod u_2$ is the memory value. But loop-unrolling affects actions as well: each iteration of $\mathtt{F}_2$, for instance, stemmed from the merger of $u_2 = 3$ subsequent native iterations. The process was as follows. Guards and increments were replaced by $\mathtt{tt}$ in every iteration of σ'', except for iterations $0 \pmod{u_2}$, where $\hat{\mathtt{B}}_2 = \mathcal{I}_0 - 4$ was used as the new guard, and for iterations $(u_2 - 1) \pmod{u_2}$, where $\mathsf{act}(\mathtt{I})[{}^{(\mathtt{X}+(u_2-1)\times\ddot{\mathtt{E}})}/_{\mathtt{X}}]$ – see Fig. 5 – was used as the new increment. In iterations $k \pmod{u_2}$, any other $\mathsf{act}(\mathtt{C})$ was replaced with $\mathsf{act}(\mathtt{C})[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}]$, and every environment ρ was updated to $\rho[\mathtt{X} := \mathsf{A}(\mathtt{X})\rho - m\mathsf{A}(\ddot{\mathtt{E}})\rho]$. After $\lfloor {}^{(\iota-\ell_2)}/_{u_2} \rfloor\, u_2$ iterations of σ, $\hat{\mathtt{B}}_2$ becomes false; thus here a new state with command $20\mathtt{g}\colon\ \neg\hat{\mathtt{B}}_2 \rightarrow 00\mathtt{g};$ was inserted. Such new states are discarded by observational abstraction $\alpha^{\mathsf{lu}}_{\mathcal{O}}$ for loop-unrolling which, for any other state $\langle \rho, \mathtt{C} \rangle$, gets rid of $\mathtt{C}$ and reverses the update of ρ using the memory value inside $\mathsf{lab}(\mathtt{C})$:

$$\alpha^{\mathsf{lu}}_{\mathcal{O}}(\mathcal{T}) \triangleq \{\alpha^{\mathsf{lu}}_{\mathcal{O}}(\sigma) \mid \sigma \in \mathcal{T}\} \qquad \alpha^{\mathsf{lu}}_{\mathcal{O}}(\sigma) \triangleq \lambda j.\ \alpha^{\mathsf{lu}}_{\mathcal{O}}(\sigma_j)$$

$$\alpha^{\mathsf{lu}}_{\mathcal{O}}(\langle \rho, im\mathtt{s}\colon\ \mathtt{A} \rightarrow im\mathtt{s}';\rangle) \triangleq \textbf{if } (\mathtt{s} = \mathtt{s}')\ \varepsilon \textbf{ else } \rho[\mathtt{X} := \mathsf{A}(\mathtt{X})\rho + m\mathsf{A}(\ddot{\mathtt{E}})\rho] \textbf{ fi } .$$

Semantic transformation t^{lu} for loop-unrolling, shown in Fig. 5, scans a trace in $\mathsf{S}(\mathtt{P})$ and unrolls any subtrace $\sigma \in \mathsf{S}(\mathtt{F})$, using factors from vectors $\mathbf{u} = \langle u_1, \ldots, u_I \rangle$ and $\boldsymbol{\ell} = \langle \ell_1, \ldots, \ell_I \rangle$. For each $u_i \geq 1, \ell_i \geq 0$ it produces unrolled `for`-loop $\mathtt{F}_i$. Native iterations left unprocessed in the rear of σ belong to $\mathtt{F} = \mathtt{F}_0$, where the equality holds since $u_0 \triangleq 1$ and $\ell_0 \triangleq 0$. Index i, initially set to 0, is ruled by function I, which increases it just at the beginning of σ and after the insertion of each new state. When the unrolling is over, I reverts i to 0. While unrolling is performed ($i > 0$), $\hat{\mathtt{B}}_i$ has to be checked and inserted every u_i iterations of σ. The count is kept through memory value m controlled by function M. The check is performed by validation function V, and it occurs whenever $m = 0$ and t^{lu} is about to transform a guard state. In particular, if $\hat{\mathtt{B}}_i$ evaluates to false, the additional state is inserted and then unrolling goes on using the next factors, if any, provided that there are still native iterations to unroll.

Proposition 3. *Let* $\mathtt{P}$ *be a program and* $\sigma \in \mathsf{S}(\mathtt{P})$*. Then* $t^{\mathsf{lu}}(\sigma)$ *is a trace and* $\alpha^{\mathsf{lu}}_{\mathcal{O}}(t^{\mathsf{lu}}(\sigma)) = \alpha^{\mathsf{lu}}_{\mathcal{O}}(\sigma)$*. Furthermore* $\mathbb{p} \circ (t^{\mathsf{lu}} \circ \mathsf{F}) = \mathbb{F}^{\mathsf{lu}} \circ \mathbb{p}$.

t^{lu} turns a trace $\sigma \in \mathsf{S}(\mathtt{F})$ into an $\alpha^{\mathsf{lu}}_{\mathcal{O}}$-equivalent trace $\sigma' \in \mathsf{S}(\mathtt{F}_1 \cup \ldots \cup \mathtt{F}_I \cup \mathtt{F})$, notwithstanding σ is a subtrace inside a trace of $\mathtt{P}$. Thanks to the fixpoint transfer, we get the algorithm yielding $\mathtt{P}' \triangleq \mathtt{P} \cup \mathtt{F}_1 \cup \ldots \cup \mathtt{F}_I$ from $\mathtt{P} \supseteq \mathtt{F}$. We express it as follows:

$\text{INIT}^{\text{lu}}(\texttt{P}) \triangleq \{\texttt{C}' \mid \exists \texttt{C} \in \texttt{P}.\ \mathsf{lab}(\texttt{C}) \in \mathfrak{L}(\texttt{P}) \ \wedge \texttt{C}' \text{ is in } t^{\text{lu}}(\texttt{C}, 0, \textbf{if } \texttt{C} \in \texttt{F} \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi})\}$

$\text{NEXT}^{\text{lu}}(\texttt{P})(\texttt{Q}) \triangleq \{\texttt{C}' \mid \exists \texttt{C} \in \texttt{P}.\ \exists \texttt{L}: \texttt{A} \rightarrow i'm'\texttt{s}';\ \in \texttt{Q}.\ \mathsf{lab}(\texttt{C}) = 00\texttt{s}' \wedge \texttt{C}' \text{ is in } t^{\text{lu}}(\texttt{C}, m', i')\}$

$\text{ITER}^{\text{lu}}(\texttt{P})(\texttt{Q}) \triangleq$ **let** $\texttt{Q}' = \texttt{Q} \cup \text{NEXT}^{\text{lu}}(\texttt{P})(\texttt{Q})$ **in if** $\texttt{Q}' = \texttt{Q}$ **then** $\texttt{Q}'$ **else** $\text{ITER}^{\text{lu}}(\texttt{P})(\texttt{Q}')$ **fi** .

5 Software Watermarking by Loop-Unrolling

In the watermarking technique we propose here, a signature is a natural number $\mathtt{s}$, which is computed iteratively in watermark variable W by mean of the following stegomark: $\texttt{W} := a$; **for** $\texttt{X} := 0$ **to** $n-1$ **do** $\texttt{W} := \xi \times \texttt{W} + b$; **od**. This stegomark implements the Horner technique for the evaluation at $x = \xi$ of n-degree polynomial $P_n(x) \triangleq ax^n + b\sum_{j=0}^{n-1} x^j$. Hence we have $\mathtt{s} = P_n(\xi)$. Let us consider an example: signature $\mathtt{s} = 120736$, obtained as the evaluation at $x = 14$ of the 3-degree polynomial $P_3(x) = -199948x^3 + 245760x^2 + 245760x + 245760$, can be computed by the following stegomark: $\texttt{W} := -199948$; **for** $\texttt{X} := 0$ **to** 2 **do** $\texttt{W} := 14{\times}\texttt{W}{+}245760$; **od**. The degree of P_n is precisely the number n of iterations performed by the for-loop in the stegomark. The stegomark is going to be embedded in a for-loop $\texttt{F} \in \mathsf{fors}(\texttt{P})$ performing, on some input $\mathcal{I}$, at least n iterations. Thus n can range from 1 to the maximum number of iterations F can perform when P is executed. Reasonably we assume that for any for-loop $\texttt{F} \in \mathsf{fors}(\texttt{P})$ there exists an input $\mathcal{I}$ such that F performs at least one iteration on $\mathcal{I}$. Thus any for-loop can be targeted for embedding. Likely, we expect that in any program which is complex enough to be worth protection there is at least a for-loop where to embed the stegomark. This because, in such programs, large amounts of data aggregate in data structures, like e.g. arrays, that need for-loops to be manipulated.

Given $\mathtt{s}$ and $n > 0$, we would like $P_n(\xi) = a\xi^n + b\sum_{j=0}^{n-1} \xi^j = \mathtt{s}$. We thereby let $a \triangleq \frac{\mathtt{s}}{\xi^n} - \frac{b}{\xi^n}\sum_{j=0}^{n-1} \xi^j$. We ask for ξ, b and a to be whole numbers, so that $\mathtt{s}$ can be safely evaluated through the stegomark. First, we require ξ^n to be a divisor of $\mathtt{s}$. In our example we have $\mathtt{s} = 120736 = 2^5 \cdot 7^3 \cdot 11$ and $n = 3$, so $\xi = 14$ is one possible choice. Next, we require b to be a nonzero multiple of ξ^n, namely $b \neq 0$ and $b = \xi^{n+n'}z$, where $n' \in \mathbb{N}$ and $z \in \mathbb{Z}$ are random numbers. In our example we set $b = 14^{3+11} \cdot 15 = 245760$. As watermarked program Q in Fig. 3 shows, ξ and b are not obfuscated. Moreover, by design, it is known that b is a multiple of ξ^n. If n' was fixed by design, e.g. $n' \triangleq 0$, then n could be easily retrieved – by just subtracting n' to the number q of times ξ divides b. This would be unpleasant because n is part of the secret watermarking key. By letting n' be selected randomly, what it is known to an attacker is that $0 < n \leq q$: the greater is n', the larger is the range of n. Programming languages do not allow numbers to exceed a prefixed maximum *MAX*. If parameters ξ and b are too big, we may compute them using ad-hoc functions fed with smaller values; this also increases the stealth of such parameters.

Embedding. In order to inlay the stegomark computing $\mathtt{s}$ in P, we run the embedding algorithm shown in Fig. 6. The algorithm looks for a for-loop F that, on a given input $\mathcal{I}$, performs at least n iterations. If the guard of F includes variables that are initialized randomly, the number of iterations on $\mathcal{I}$ may not be fixed. Therefore we let ι be the minimum number of iterations of F on $\mathcal{I}$, and we require $\iota \geq n$. Furthermore, stegomark

funct *embed*$(\mathtt{P}, \mathcal{I}, \mathtt{W}, n, \xi, a, b)$
$\mathtt{P}' \leftarrow \mathtt{P}$; $\mathcal{F} \leftarrow \mathsf{fors}(\mathtt{P})$;
while $\mathcal{F} \neq \emptyset \wedge \mathtt{P}' = \mathtt{P}$ **do**
$\mathtt{F} \leftarrow$ *next*$(\mathcal{F})$; (by def. $\mathtt{F} \triangleq \{\bar{\mathtt{G}}, \mathtt{G}, \mathtt{I}\} \cup \mathtt{H}$)
$\mathcal{F} \leftarrow \mathcal{F} \setminus \{\mathtt{F}\}$;
$\iota \leftarrow$ min # iterations of $\mathtt{F}$ when $\mathtt{P}$ is run on $\mathcal{I}$;
if ($\iota \geq n \wedge \mathtt{W}$ is dead wrt. the guard of $\mathtt{F}$)
$\mathbf{u} \leftarrow \langle \iota \rangle$; $\boldsymbol{\ell} \leftarrow \langle 0 \rangle$;
$\mathtt{Q} \leftarrow \mathbb{t}^{\mathrm{lu}}(\mathtt{P};\ \mathtt{F}, \mathbf{u}, \boldsymbol{\ell})$;
if (there exists $\mathtt{C} \in \mathtt{Q}$ such that
$\mathtt{C} = \mathtt{L}\texttt{:}\ \mathtt{A} \rightarrow \mathtt{L}'\texttt{;}$
$\mathtt{L} = 1\delta\mathtt{s}$ with $\mathtt{s} \neq \mathtt{i} \wedge \delta \in [0, \iota - n]$
$\mathtt{A} = \mathtt{Y} := \mathtt{E}$ with $\mathtt{Y} \in \mathsf{var}(\mathtt{Q}) \wedge \mathtt{E} \in \mathbb{E}$
$\mathtt{L}' = 1\delta\mathtt{v}$ with $\mathtt{v} \in \mathsf{lab}(\mathtt{F})$)
$\mathtt{S} \leftarrow$ *slice*$(\mathtt{Q}, \mathtt{C})$;
$y \leftarrow$ value of $\mathtt{Y}$ when $\mathtt{S}$ is run on $\mathcal{I}$;
$r_0 \leftarrow$ a random number in $\mathbb{Z} \setminus \{y\}$;
$r_1 \leftarrow$ a random number in $\mathbb{Z}$;
let $f(\mathtt{Y}) \triangleq \dfrac{r_1 - a}{r_0 - y}(\mathtt{Y} - y) + a$;
$\mathtt{w} \leftarrow$ a label from $\mathsf{lab}(\mathtt{H} \cup \{\mathtt{I}\})$ that
is reached after $\mathtt{v}$ in the CFG of $\mathtt{P}$;
$\theta_0 \leftarrow \langle \mathtt{v}, \mathtt{W}, f(\mathtt{Y}) \rangle$;
$\theta_1 \leftarrow \langle \mathtt{w}, \mathtt{W}, \xi \times \mathtt{W} + b \rangle$;
$\mathtt{P}' \leftarrow \mathbb{t}^{\mathrm{in}}(\mathtt{P};\ \theta_0, \theta_1)$; **fi fi od**
return $\langle \mathtt{P}', \langle \mathcal{I}, \delta, n \rangle \rangle$;

funct *extract*$(\mathtt{P}', \langle \mathcal{I}, \delta, n \rangle)$
$\mathcal{S} \leftarrow \emptyset$;
for each $\mathtt{F} \in \mathsf{fors}(\mathtt{P}')$ **do**
$\iota \leftarrow$ # iterations of $\mathtt{F}$ when $\mathtt{P}'$ is run on $\mathcal{I}$;
if $(\iota \geq n)$
$\mathbf{u} \leftarrow \langle 1, n \rangle$;
$\boldsymbol{\ell} \leftarrow \langle \iota - \delta, \iota - \delta - n \rangle$;
$\mathtt{Q} \leftarrow \mathbb{t}^{\mathrm{lu}}(\mathtt{P}';\ \mathtt{F}, \mathbf{u}, \boldsymbol{\ell})$;
for each $\mathtt{L}\texttt{:}\ \mathtt{A} \rightarrow \mathtt{L}'\texttt{;} \in \mathtt{Q}$ such that
$\mathtt{L} = 20\mathtt{s}$ with $\mathtt{s} \in \mathbb{S}$
$\mathtt{A} = \mathtt{W} := \mathtt{E}$ with $\mathtt{W} \in \mathsf{var}(\mathtt{Q}) \setminus \mathsf{var}(\mathtt{E}) \wedge \mathtt{E} \in \mathbb{E}$
$\mathtt{L}' = 20\mathtt{s}'$ with $\mathtt{s} \in \mathbb{S}$
do
$\mathtt{Q}' \leftarrow \mathtt{Q} \setminus \{\mathtt{C} \in \mathtt{Q} \mid \exists m > 0.\ \mathsf{lab}(\mathtt{C}) = 2m\mathtt{s}\}$;
$\mathtt{R} \leftarrow \{\mathtt{L}''\texttt{:}\ \mathtt{A}' \rightarrow \mathtt{L}'''\texttt{;} \in \mathtt{Q}' \mid$
$\mathtt{L}'' = 2[n-1]\mathtt{s}''$ with $\mathtt{s}'' \in \mathbb{S}$
$\mathtt{A}' = \mathtt{W} := \mathtt{E}'$ with $\mathtt{W} \in \mathsf{var}(\mathtt{E}) \wedge' \mathtt{E} \in \mathbb{E}$
$\mathtt{L}''' = 2[n-1]\mathtt{s}'''$ with $\mathtt{s}''' \in \mathbb{S}\}$
if ($\mathtt{R}$ is a singleton with element $\mathtt{C}$)
$\mathtt{S} \leftarrow$ *slice*$(\mathtt{Q}', \mathtt{C})$;
$w \leftarrow$ value of $\mathtt{W}$ when $\mathtt{S}$ is run on $\mathcal{I}$;
$\mathcal{S} \leftarrow \mathcal{S} \cup \{w\}$; **fi od fi od**
return $\mathcal{S}$;

Fig. 6. Embedding and extraction algorithms

variable $\mathtt{W}$ must be dead during the execution of $\mathtt{F}$. If such a `for`-loop does not exists in $\mathtt{P}$, the algorithm fails and returns $\mathtt{P}$ and the empty key. Otherwise, it gets from $\mathtt{F}$ an unrolled `for`-loop $\mathtt{F}_1$ which syntactically displays all the ι iterations as sequential code: actually, any command $\mathtt{C}' \in \mathtt{F}_1$ derived from iteration $m \in [0, \iota)$ is such that $\exists \mathtt{s} \in \mathbb{S}.\ \mathsf{lab}(\mathtt{C}') = 1m\mathtt{s}$; here we also say that $\mathtt{C}'$ is at *offset* m. Next, the algorithm looks for a command $\mathtt{C}$ at offset $\delta \in [0, \iota - n)$ such that $\mathsf{act}(\mathtt{C}) = \mathtt{Y} := \mathtt{E}$. If it succeeds, it computes actual value y of $\mathtt{Y}$ on input $\mathcal{I}$, using backward-slicing with criterion $\mathtt{C}$. Then it lets first-degree polynomial $f(\mathtt{Y})$ to model the line passing through points (y, a) and (r_0, r_1) in the Cartesian coordinate system; in such a way, one possible dependence between y and parameter a of the stegomark is established. Finally, the algorithm comes back to subject program $\mathtt{P}$, and it inserts $\mathtt{W} := f(\mathtt{Y})$ at entrypoint $\mathsf{lab}(\mathtt{C})$ and $\mathtt{W} := \xi \times \mathtt{W} + b$ somewhere below, inside the body of $\mathtt{F}$. In such a way, it obtains marked program $\mathtt{P}'$ which it returns together with key $\langle \mathcal{I}, \delta, n \rangle$. Note that we can guarantee $f(\mathtt{Y}) = a$ only at offset δ. If $\mathtt{Y}$ denotes stochastic behavior, i.e., it changes its own value from one iteration to another, the knowledge of δ becomes essential at extraction time to get the correct initialization of $\mathtt{W}$. This improves reliability and stealth of the watermark. The iteration at offset δ is the *promoter* of the signature recovery, and δ measures its displacement in the sequence of the ι iterations.

As shown in Fig. 3, the embedding phase basically consists in a pair of assignment insertions. To inlay in P our signature $\mathtt{s} = P_3(14) = 120736$, we want the for-loop to perform at least $n = 3$ iterations, so we let $\mathcal{I}_0 \triangleq 8$, obtaining $\iota = 8$. Furthermore, by fixing $\mathcal{I}_1 \triangleq 13$ and $f \triangleq \lambda \mathtt{Y}.\ (215 - \mathtt{Y}) \times 259$, we ensure that, at entry point v of the iteration at offset $\delta = 3$, we have $y = \mathsf{Fib}(\mathtt{X} + \mathcal{I}_1) = \mathsf{Fib}(3 + 13) = 987$ and $f(987) = -199948 = a$. Thus, once we have chosen label $\mathtt{w} \equiv \mathtt{i}$ as target entry point for the second assignment, we insert $\mathtt{W} := f(\mathtt{Y})$ at v and $\mathtt{W} := 14 \times \mathtt{W} + 245760$ at w.

Extraction. To extract our signature s from marked program $\mathtt{P}'$, we need to deliver $\mathtt{P}'$ and key $\kappa = \langle \mathcal{I}, \delta, n \rangle$ to the algorithm described in Fig. 6. From each for-loop F in $\mathtt{P}'$ performing on input $\mathcal{I}$ a number $\iota \geq n$ of iterations, the algorithm tries to gain a set of candidate signatures; the final result of the extraction is a union set $\mathcal{S}$ collecting altogether the candidate signatures coming from each set. To gain a set of candidate signatures from F, the algorithm unrolls F into $\mathtt{F}_1 \cup \mathtt{F}_2 \cup \mathtt{F}$. The unrolling is instrumented so as to make unfold, within the body of $\mathtt{F}_2$, only the n iterations at offsets from δ to $\delta + n - 1$. The iterations at lesser or greater offsets are left folded in $\mathtt{F}_1$ and F respectively. Iteration at δ, now denoting offset 0 within the body of $\mathtt{F}_2$, is potentially a promoter. In particular, any of its assignment $\mathtt{W} := \mathtt{E}$ not defining W in terms of itself may be the initializer of the stegomark. Given such an assignment command C, the algorithm removes its copies at nonzero offsets within $\mathtt{F}_2$. Next, at offset $n - 1$, it looks for a unique assignment $\mathtt{C}'$ redefining W it terms of itself, and it applies backward-slicing using $\mathtt{C}'$ as criterion. The result is a program S which on input $\mathcal{I}$ first provides an initialization to W, then updates it n times, thus computing candidate signature w. In particular, if W is the watermark variable, then $w = \mathtt{s}$. Once identified s among the candidates in $\mathcal{S}$, one has only to prove it to be his/her signature, as discussed above.

We now exploit the algorithm to extract signature $\mathtt{s} = 120736$ from watermarked program $\mathtt{P}'$ of Fig. 3. Recall that in our running example the key κ is $\langle\langle \mathcal{I}_0, \mathcal{I}_1 \rangle, \delta, n\rangle = \langle\langle 8, 13\rangle, 3, 3\rangle$ and $\iota = 8$. After the unrolling of $\mathtt{F} \subseteq \mathtt{P}'$, we get program Q shown in Fig. 3. The promoter always covers entry points $20\mathtt{s}$, with $\mathtt{s} \in \mathbb{S}$. Here both variable Y and variable W might initialize the stegomark. However, only W at entry point $22\underline{\mathtt{i}}$ is able to update itself. After the slicing, we get a program $\mathtt{S} \subseteq \mathtt{Q}$ which, on input $\langle 8, 13\rangle$, sets X to 3, Y to $\mathsf{Fib}(3 + 13) = 987$ and W to $(215 - 987) \times 259 = -199948 = a$; just before terminating, S updates $n = 3$ times W, finally getting $w = 120736 = \mathtt{s}$.

6 Discussion

In this paper we exploit the semantics of for-loops to hide watermarks. Loop iterations are described extensionally by traces of execution in which iterations come one after another. When abstracted to code, they collapse into a unique loop body. Thus embedding the stegomark in the loop body means embedding it in every iteration. Our idea is to set up the stegomark so that only one iteration, the promoter, can provide the correct initialization for the computation of the signature. The choice and the localization of the promoter take place automatically thanks to loop-unrolling, used as a transformation which abstract iterations from trace to code without making them collapse. We think that our watermarking technique may be extended to other programming constructs which, like for-loops, provide code reuse, such as recursive functions and objects.

Signature $\mathfrak{s}$ must reliably identify the author of the watermarked program. To this end, the author can let $\mathfrak{s}$ be the product of a set of prime numbers. If some factors of $\mathfrak{s}$ are large enough, its factorization is computationally unfeasible, yet the author is able to produce it. False positives may be obtained at extraction time, both in the case of marked and unmarked loops. However, it is unlikely that their factorization is computationally unfeasible and yet known by a malicious claimer. If the extraction of the signature $\mathfrak{s}$ results in a overflow runtime error then, as suggested in [13], $\mathfrak{s}$ can be replaced with an equivalent set of smaller signatures obtained through the Chinese remainder theorem.

Watermarked programs can include more than one signature. However, they do not record which signature was inserted first, and which ones were inserted later through *additive attacks*. Unfortunately, our watermarking technique does not provide any means to register temporal precedence of signatures. To the best of our knowledge, vulnerability to additive attacks is a common drawback to all the exiting watermarking techniques [5,4]. This key problem might be solved if the insertion of the signature coincided with a not reversible semantics-preserving program evolution [3]: in such a case the order of insertion of signatures would become relevant, especially if later evolutions were strictly dependent on earlier ones. As in the field of code obfuscation [7], nontrivial semantics-preserving program transformations are likely to be systematically derived only from semantics-based frameworks. Consequently, we suppose that a better exploitation of the gap between semantics and syntax may be of help in the design of watermarking techniques that can withstand additive attacks.

Typical loop transformations [2], such as loop-reversal, loop-unrolling and loop-blocking, might distort the syntactic structure of the marked loop and obstruct the extraction of the signature; however, they are applicable only when the number of iterations can be ultimately quantified; thus a countermeasure is to embed the watermark in a `for`-loop not enjoying this property, e.g. a `for`-loop that updates an array of arbitrary length. To avoid that the inserted assigments are declared useless for the output, we *must* introduce fake dependencies between the output and `W`, for example by using opaque predicates which require hard program analyses to be removed [8]. Indeed our technique does not provide innovative contribution to the age-old problem of the resilience of watermarks. Anyway we think that semantics-based approaches may help us understand to which extent watermarks can be tied to the very core of programs.

As suggested by Fig. 1, our watermarking technique seems to resemble the DNA transcription step in protein biosynthesis. During transcription, information coded in a DNA stretch is extracted and recoded in a complementary RNA molecule. In particular, DNA unwinds and produces a small open stretch containing a promoter, which is a regulatory region providing an entry point for transcription. The transcribed RNA molecule can be partitioned in exons/introns, i.e., subregions carrying useful/useless information. Through splicing, every intron in RNA is discarded to keep only exons. Now, notice that the marked loop can be seen as the folded DNA: at extraction time, partially unrolling the marked loop corresponds to partially unwinding DNA and producing a stretch; the iteration targeted by δ is the promoter; slicing and the other minor removals correspond to RNA splicing. The idea of inserting proprietary information in a DNA molecule has been initially explored in [18]. Surely, our technique is not applicable to DNA. However, this comparison could provide intriguing insights for further research.

References

1. Apt, K.R., Plotkin, G.D.: Countable nondeterminism and random assignment. J. ACM 33(4), 724–767 (1986)
2. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. 26(4), 345–420 (1994)
3. Cohen, F.B.: Operating system protection through program evolution. Comput. Secur. 12(6), 565–584 (1993)
4. Collberg, C., Carter, E., Debray, S., Huntwork, A., Kececioglu, J., Linn, C., Stepp, M.: Dynamic path-based software watermarking. SIGPLAN Not 39(6), 107–118 (2004)
5. Collberg, C., Thomborson, C.: Software watermarking: Models and dynamic embeddings. In: Principles of Programming Languages 1999, POPL 1999, San Antonio, TX (January 1999)
6. Collberg, C., Thomborson, C.: Watermarking, tamper-proofing, and obfuscation – tools for software protection. Technical Report TR00-03, University of Arizona (February 10, 2000)
7. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland (July 1997)
8. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Principles of Programming Languages 1998, San Diego, CA (1998)
9. Cousot, P.: Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. Theoretical Computer Science 277(1-2), 47–103 (2002)
10. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, pp. 238–252. ACM Press, New York (1977)
11. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, pp. 269–282. ACM Press, New York (1979)
12. Cousot, P., Cousot, R.: Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In: Conference Record of the 19th ACM Symposium on Principles of Programming Languages, pp. 178–190. ACM Press, New York (2002)
13. Cousot, P., Cousot, R.: An abstract interpretation-based framework for software watermarking. In: Conference Record of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, Italy, ACM Press, New York (2004)
14. Davidson, R.L., Myhrvold, N.: Method and systems for generating and auditing a signature for a computer program. US patent 5.559.884, Assignee: Microsoft Corporation (1996)
15. Moskowitz, S.A., Cooperman, M.: Method for stega-cipher protection of computer code. US patent 5.745.569, Assignee: The Dice Company (1996)
16. Nagra, J., Thomborson, C.D.: Threading software watermarks. In: Fridrich, J. (ed.) IH 2004. LNCS, vol. 3200, pp. 208–223. Springer, Heidelberg (2004)
17. Qu, G., Potkonjak, M.: Hiding signatures in graph coloring solutions. In: Pfitzmann, A. (ed.) IH 1999. LNCS, vol. 1768, pp. 348–367. Springer, Heidelberg (2000)
18. Shimanovsky, B., Feng, J., Potkonjak, M.: Hiding data in Dna. In: Revised Papers from the 5th International Workshop on Information Hiding, London, UK. Springer, Heidelberg (2003)
19. Weiser, M.: Program slicing. In: ICSE 1981: Proceedings of the 5th international conference on Software engineering, Piscataway, NJ, USA, pp. 439–449. IEEE Press, Los Alamitos (1981)