# An Abstract Interpretation-Based Framework for Software Watermarking

Patrick COUSOT and Radhia COUSOT

École Normale Supérieure
75230 Paris cedex 05, France
Patrick.Cousot@ens.fr

CNRS & École Polytechnique
91128 Palaiseau cedex, France
Radhia.Cousot@polytechnique.fr

## Abstract

Software watermarking consists in the intentional embedding of indelible stegosignatures or watermarks into the subject software and extraction of the stegosignatures embedded in the stegoprograms for purposes such as intellectual property protection. We introduce the novel concept of *abstract software watermarking*. The basic idea is that the watermark is hidden in the program code in such a way that it can only be extracted by an abstract interpretation of the (maybe non-standard) concrete semantics of this code. This static analysis-based approach allows the watermark to be recovered even if only a small part of the program code is present and does not even need that code to be executed. We illustrate the technique by a simple abstract watermarking protocol for methods of Java™ classes. The concept applies equally well to any other kind of software (including hardware originally specified by software).

**Categories and Subject Descriptors:** D.2.9 Software Engineering/Management: Copyrights.

**General Terms:** Algorithms, Reliability, Security, Languages, Theory, Legal Aspects, Verification.

**Keywords:** Abstract Interpretation, Authentication, Copyrights Protection, Fingerprinting, Identification, Intellectual Property Protection, Obfuscation, Software Authorship, Software Watermarking, Static Analysis, Steganography, Stegoanalyst, Stegoattacks, Stegokey, Stegomark, Stegosignature, Tamper-proofing, Trustworthiness, Validation Watermarking.

## 1. Introduction

Digital information hiding techniques such as steganography, digital watermarking and fingerprinting have received much attention from the research community and industry. With few notable exceptions [4, 15, 16], relatively little work has been done on *software watermarking* that consists in *embedding* (that is the indelible unobtrusive fixing of invisible *stegosignatures*[1] or *watermarks*, such as cryptographic signature and timestamp, in subject programs) and *extraction* (that is the detecting) of the stegosignatures) embedded in the *stegoprograms* (that is watermarked program sources).

[1] "stego-xxx" means "xxx" in the context of hiding some embedded secret information.


*POPL'04,* January 14–16, 2004, Venice, Italy.


Digital watermarking is to be contrasted with public-key encryption, which also transforms subject files into another form so that they become unusable without the decryption key. Once the software is decrypted, it is free of any residual effect of encryption. Hence, licit use can only be ensured by keeping a list of certified customers of the decrypted software, by using licensing information, by hardware monitoring of the program execution or by linking the software to the hardware of a specific machine or to a specific movable piece of hardware such as a dongle [4, 16] or to a specific user public key. An alternative would be to use "encrypted execution" so that code never executes in-the-clear thus requiring a customized encrypted execution mechanism. These techniques are inflexible (e.g. for software to be downloaded from a web page).

Unlike encryption, digital watermarking leaves the subject object basically intact, recognizable and usable. Further, whereas decrypted documents are free of any residual effects of encryption, digital watermarks are on the contrary designed to be persistent in viewing, printing, or subsequent re-transmission or dissemination [10]. Watermarking does not aim so much to stop illicit use, but to prove ownership of the software and the algorithms used in the software [16]. Notice that in general the watermark may hide an encrypted signature, a problem that we handle by taking the signature to be an arbitrary number within an arbitrary large bound.

We illustrate our approach on the watermarking of methods in Java™ classes. Complete sophisticated Java™ applications can be easily constructed from sets of Java™ classes including reusable methods with high proprietary content (such as licence control code or original algorithms) which one might want to watermark. Java™ classes are easy to decompile and reverse engineer since they are distributed in hardware independent virtual machine code retaining almost all the information of the original Java™ code. Obfuscators and code transformers can be used to make reverse engineering more difficult [6] but also to hide the origin of Java™ classes. In this case indelible watermarking is an appealing complementary solution.

Two types of digital watermarks may be distinguished. *Visible watermarks* (such as a logo placed in a corner of the screen image by the TV industry) discourage the illicit use by immediately claiming ownership. *Invisible watermarks*, on the other hand, are potentially useful as a means of identifying the source, author, creator, owner, distributor or authorized consumer of a multimedia object (image, audio, video, text, etc). For this purpose, the objective is to permanently and unalterably mark the object. Since the watermark is not visible, a special detection software is needed to extract the original signature. In the event of illicit use, the watermark would facilitate the claim of ownership.

According to Collberg and Thomborson's informal taxonomy of software watermarking techniques [3], one can distinguish between static and dynamic watermarking.

• ***Static watermarking*** stores signatures in the program source either as data (e.g. an image, a string, etc) or code (e.g. in the code control structure). So, the stegosignature can be extracted from the text of the program (or the program syntax), without any need for execution.

• ***Dynamic watermarking*** stores signatures in the program execution state and so requires the program to be executed in order to extract the stegosignature. [3] distinguishes between:

*Easter egg watermarking* stores signatures in a piece of code executed for a highly unusual input to the application.

*Dynamic data structure watermarking* stores signatures in the program data if and when executed with particular inputs. The signature is exhibited by a watermark extraction routine examining these signed program data.

*Dynamic execution trace watermarking* stores signatures in the program execution trace when executed with special inputs. The extraction consists in recognizing specific properties of the addresses or operations on the trace.

One difficulty with dynamic watermarking is that the special input revealing the watermark can be localized by monitoring program execution using standard instrumentation techniques and removed by debugging techniques in which case it must be considered ineffective. Dynamic data structure watermarking can be obliterated by program transformations such as variable splitting/merging while dynamic execution trace watermarking can be erased by program optimization and transformation. This pessimistic view dismisses all current and future dynamic watermarking schemes as completely ineffective.

• ***Abstract watermarking*** is the new framework that we introduce for software watermarking. Abstract watermarking is fully automatic and different from both static and dynamic watermarking.

Abstract watermarking is a static watermarking in that the extraction of the signature requires no execution of the program. It follows that no easily recognized special input is needed for extraction.

Abstract watermarking is a dynamic watermarking in that the stegosignature is hidden in the concrete semantics of the program (which may be a non-standard one). However execution of the stegoprogram (or just of the stegomark inserted in the subject program) will not reveal the stegosignature.

Abstract watermarking is different from existing static or dynamic watermarking methods in that the stegosignature extraction is by static analysis of the stegoprogram that is by abstract interpretation of its (non-standard) concrete semantics (whence the "abstract" qualifier of this watermarking method). Moreover the abstract extractor can be parametrised by a stegokey to fix the specific abstract domain used for the signature extraction. Enriching the terminology of [15], this is a *semi-visible* watermarking in that even if the presence of a stegomark may be visible (e.g. by making public the watermark embedding algorithm), the watermark or signature will be invisible to those not owning the detection software and the corresponding secret stegokey.

# 2. Principle of Software Watermarking

We let $\mathcal{P}$ be the set of considered programs and $\mathcal{S}$ be the set of stegosignatures. The watermark embedder is $\mathcal{W} \in \mathcal{P} \mapsto \mathcal{S} \mapsto \mathcal{P}$. The watermark extractor is $\mathcal{E} \in \mathcal{P} \mapsto \wp(\mathcal{S})$. The principle of software watermarking/fingerprinting is that the stegosignature can be extracted from the stegoprograms, that is for all $s \in \mathcal{S}$ and $P \in \mathcal{P}$, $s \in \mathcal{E}[\![\mathcal{W}[\![P]\!](s)]\!]$. (Following the conventions of denotational semantics for the application, syntactic arguments are between double square brackets $[\![...]\!]$ while semantic/mathematical arguments are between round brackers (...).)

The stegomark approach to software watermarking first consists in defining a *stegomarker* $\mathcal{M}_r \in \mathcal{S} \mapsto \mathcal{P}$ to encode the signature $s$ into a program $\mathcal{M}_r(s)$ called the *stegomark*. Second, a *stegoinlayer* $\mathcal{I} \in (\mathcal{P} \times \mathcal{P}) \mapsto \mathcal{P}$ is used to compose the stegomark $\mathcal{M}_r(s)$ with the subject program $P \in \mathcal{P}$ to get the watermarked stegoprogram: $\mathcal{W}[\![P]\!](s) \triangleq \mathcal{I}[\![P, \mathcal{M}_r(s)]\!]$.

# 3. Abstract Software Watermarking

In abstract software watermarking, the signature is hidden in the semantics of the stegomark embedded in the subject program, whence in the semantics of the stegoprogram. The signature extraction is by static analysis of the stegoprogram using some *abstract interpretation* of the stegoprogram semantics.

## 3.1 Concrete Semantics

Since the stegosignature is hidden in the program semantics, we must define a collecting semantics [7], at some level of abstraction such as sets of traces, sets of reachable states, etc. In this application, we consider the set of reachable states that is the set of possible descendants of the initial states during program execution.

Formally, the small-step operational semantics of the program $P$ is a triple $\langle \Sigma[\![P]\!], \mathcal{I}[\![P]\!], \tau[\![P]\!] \rangle$ where $\Sigma[\![P]\!]$ is a set of (memory and control) states, $\mathcal{I}[\![P]\!] \subseteq \Sigma[\![P]\!]$ is the set of initial states and $\tau[\![P]\!] \subseteq \Sigma[\![P]\!] \times \Sigma[\![P]\!]$ is the transition relation relating a state $s$ to a possible successor $s'$ whenever $\langle s, s' \rangle \in \tau[\![P]\!]$. The semantics $\mathcal{S}[\![P]\!]$ of the program $P$ is the set of execution traces generated by $\langle \Sigma[\![P]\!], \mathcal{I}[\![P]\!], \tau[\![P]\!] \rangle$. We consider the reachable states abstraction of this semantics [7].

Formally, given a set $S$ and relations $t$, $t' \subseteq S \times S$, we let $t^0 \triangleq \{\langle s, s \rangle \mid s \in S\}$ be the identity, $t \circ t' \triangleq \{\langle s, s'' \rangle \mid \exists s' \in S : \langle s, s' \rangle \in t \wedge \langle s', s'' \rangle\} \in t'\}$ be the composition of relations $t$ and $t'$, $t^{n+1} \triangleq t \circ t^n$ be the powers of $t$, $t^* \triangleq \bigcup_{n \geq 0} t^n$ be the transitive reflexive closure of $t$ and $post[t](X) \triangleq \{s' \mid \exists s \in X : \langle s, s' \rangle \in t\}$ be the right image of $X$ by $t$ (also called the strongest post condition of $X$ for $t$). The set of reachable states of program $P$ is:

$$\mathcal{R}[\![P]\!] \triangleq post[\tau[\![P]\!]^*](\mathcal{I}[\![P]\!]) = \mathrm{lfp}^{\subseteq} \mathcal{F}[\![P]\!] = \bigcup_{n \geq 0} \left(\mathcal{F}[\![P]\!]\right)^n (\emptyset)$$

where $\mathcal{F}[\![P]\!](X) \triangleq \mathcal{I}[\![P]\!] \cup post[\tau[\![P]\!]](X)$ .

and $\mathrm{lfp}^{\leq} f$ is the $\leq$-least fixpoint of $f$ (the existence of which is guaranteed above by Tarski's fixpoint theorem). $\mathcal{R}[\![P]\!]$ is, generally, not computable.

## 3.2 Abstract Semantics

Using abstract interpretation, concrete program properties (that is subsets of $\Sigma[\![P]\!]$) are approximated in an *abstract domain* $L[\![P]\!]$

through an abstraction function $\alpha$ and a concretization function $\gamma$ chosen such that we have the Galois connection [7]:

$$\langle \wp(\Sigma[\![P]\!]), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle L[\![P]\!], \sqsubseteq \rangle \ . \tag{1}$$

By definition of Galois connections, this means that $\langle \wp(\Sigma[\![P]\!]), \subseteq \rangle$ is a partially ordered set (it is indeed a complete lattice), $\langle L[\![P]\!], \sqsubseteq \rangle$ is also a partially ordered set and for all $X \subseteq \Sigma[\![P]\!]$ and $\overline{X} \in L[\![P]\!]$:

$$\alpha(X) \sqsubseteq \overline{X} \iff X \subseteq \gamma(\overline{X}) \ .$$

Given an abstract transformer $\overline{\mathcal{F}}_\alpha[\![P]\!]$ such that for all $\overline{X} \in L[\![P]\!]$:

$$\mathcal{F}[\![P]\!] \circ \gamma(\overline{X}) \quad \subseteq \quad \gamma \circ \overline{\mathcal{F}}_\alpha[\![P]\!](\overline{X})$$

the abstract semantics is defined as:

$$\overline{\mathcal{R}}_\alpha[\![P]\!] \quad \triangleq \quad \mathrm{lfp}^{\sqsubseteq} \overline{\mathcal{F}}_\alpha[\![P]\!] \ . \tag{2}$$

If $\langle L[\![P]\!], \sqsubseteq \rangle$ is computer representable and satisfies the ascending chain condition then $\overline{\mathcal{R}}_\alpha[\![P]\!]$ is computable iteratively as:

$$\overline{\mathcal{R}}_\alpha[\![P]\!] \quad = \quad \bigsqcup_{n \geq 0} \left(\overline{\mathcal{F}}_\alpha[\![P]\!]\right)^n (\bot) \tag{3}$$

where $\bot \triangleq \alpha(\emptyset)$ is the infimum and the join

$$\bigsqcup \overline{X} \triangleq \alpha(\bigcup_{\overline{x} \in \overline{X}} \gamma(\overline{x}))$$

abstracts set union. The abstract semantics $\overline{\mathcal{R}}_\alpha[\![P]\!]$ is an approximation of the reachable states of program $P$ in that:

$$\mathcal{R}[\![P]\!] \quad \subseteq \quad \gamma(\overline{\mathcal{R}}_\alpha[\![P]\!]) \ . \tag{4}$$

A static analyser is a terminating program, written for a given abstraction $\alpha$, which, given a program $P$ as input, outputs $\overline{\mathcal{R}}_\alpha[\![P]\!]$.

The static analyser can be made parametric in a computer representation of the abstract domain $\langle L[\![P]\!], \sqsubseteq, \bot, \sqcup \rangle$ and the abstract transformer $\overline{\mathcal{F}}_\alpha[\![P]\!]$ so that e.g. the fixpoint computation:

$$\mathrm{lfp}^{\sqsubseteq} \overline{\mathcal{F}}_\alpha[\![P]\!] \quad = \quad \bigsqcup_{n \geq 0} \left(\overline{\mathcal{F}}_\alpha[\![P]\!]\right)^n (\bot)$$

can be generated automatically.

Static analyser generators contain generic abstractions $\alpha$ which are predefined in terms of simpler ones. For a simple example, let us consider that the states $\langle c, m \rangle \in \Sigma[\![P]\!] = \mathcal{C}[\![P]\!] \times \mathcal{M}[\![P]\!]$ are made up of a control state $c \in \mathcal{C}[\![P]\!]$, and a memory state $m \in \mathcal{M}[\![P]\!]$ assigning values $m(\mathtt{X}) \in \mathcal{V}[\![P]\!]$ to program variables $\mathtt{X} \in \mathbb{X}[\![P]\!]$ so that $\mathcal{M}[\![P]\!] \triangleq \mathbb{X}[\![P]\!] \mapsto \mathcal{V}[\![P]\!]$. Assume that we are interested in analyses recording information for a subset $V$ of the program variables only (for example $V$ might be the set of variables of a given type, such as the integer variables). Static analyser generators such as SOOT [20] provide a non-relational abstraction as a predefined abstract domain $L[\![P]\!]$ that is parametrised by an abstract domain $D$ abstracting the values of variables:

$$\langle \wp(\mathcal{V}[\![P]\!]), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle D, \sqsubseteq \rangle \ .$$

Then, we have $\langle \wp(\Sigma[\![P]\!]), \subseteq \rangle \underset{\alpha_{P,V}}{\overset{\gamma_{P,V}}{\leftrightarrows}} \langle L[\![P]\!], \sqsubseteq \rangle$ by defining the global abstraction $\alpha_{P,V}(X)$ of $X \subseteq \Sigma[\![P]\!]$ as:

$$\alpha_{P,V}(X) \triangleq \prod_{c \in \mathcal{C}[\![P]\!]} \prod_{\mathtt{X} \in V} \alpha(\{m(\mathtt{X}) \mid \langle c, m \rangle \in X\}) \ .$$

This global abstraction $\alpha_{P,V}$, which we implicitly use in the following, collects for each program point $c$ and each variable $\mathtt{X} \in V$ we are interested in, an abstraction, by the elementary $\alpha$, of the set of values of this variable (which is reachable in some execution of $P$ from the initial states).

The abstract transformer $\overline{\mathcal{F}}_\alpha[\![P]\!]$ can then be predefined in terms of basic abstract operations of the language $\mathcal{P}$. The abstraction $\odot$ of a $k$-ary concrete operation $\cdot$ (with domain $\mathrm{dom}(\cdot)$ and codomain $\mathrm{codom}(\cdot)$) should satisfy for all $\overline{x}_1 \in L[\![P]\!], \ldots, \overline{x}_k \in L[\![P]\!]$ [7]:

$$\begin{array}{rl} \{\cdot(x_1, \ldots, x_k) \mid x_1 \in \gamma(\overline{x}_1) \wedge \ldots \wedge x_k \in \gamma(\overline{x}_k) \wedge \\ \qquad (x_1, \ldots, x_k) \in \mathrm{dom}(\cdot)\} & \subseteq \quad \gamma(\odot(\overline{x}_1, \ldots, \overline{x}_k)) \ . \end{array}$$

## 3.3 *Abstract* Software Watermarking

The principle of abstract software watermarking is to choose a particular abstraction $\alpha$ as well as an *abstract extractor* $\overline{\mathcal{E}}(Q) \in L[\![Q]\!] \mapsto \mathcal{S}$ to extract the stegosignatures of a program $Q \in \mathcal{P}$:

$$\mathcal{E}[\![Q]\!] \ = \ \overline{\mathcal{E}}(\overline{\mathcal{R}}_\alpha[\![Q]\!]) \ . \tag{5}$$

The correctness conditions of Sec. 2 must be satisfied, so:

$$s \ \in \ \overline{\mathcal{E}}(\overline{\mathcal{R}}_\alpha[\![\mathcal{W}[\![P]\!](s)]\!]) \ .$$

Informally stated, we can watermark the subject program $P$ with stegosignature $s$ by transformation of the program $P$ into the stegoprogram $\mathcal{W}[\![P]\!](s)$ where $s$ is invisible. The signature can be extracted from the stegoprogram $\mathcal{W}[\![P]\!](s)$ by extraction with $\overline{\mathcal{E}}$ from its abstract semantics $\overline{\mathcal{R}}_\alpha[\![\mathcal{W}[\![P]\!](s)]\!]$ which is entirely specified by the abstraction $\alpha$. By maintaining $\alpha$ and $\overline{\mathcal{E}}$ secret, it is computationally hard, if not impossible to extract the stegosignature.

The method is not static in that the stegosignature is not directly hidden in the program syntax that is the data or the control structure of the stegoprogram $\mathcal{W}[\![P]\!](s)$ nor dynamic since it is not hidden in the concrete semantics $\mathcal{R}[\![\mathcal{W}[\![P]\!](s)]\!]$ of the stegoprogram (which need not be executed to extract the signature from an execution trace). It is *abstract*, that is hidden in the abstract semantics $\overline{\mathcal{R}}_\alpha[\![\mathcal{W}[\![P]\!](s)]\!]$ of the stegoprogram, which is computable by static analysis of this stegoprogram.

In particular, the concrete semantics which is used, need not be the collecting semantics for the standard semantics as considered up to now. For example, if we want to watermark a method $M$ of a program $P$, we can use the semantics of $M$ where the parameters and global variables are all unknown as in a so-called monovariant intraprocedural static analysis. The semantics of $M$ can also be chosen to be non-standard and kept as an additional secret. A simple example would be a non-standard interpretation of floating point constants and operations as integer constants and operations.

## 3.4 Principle of *Parametrised* Abstract Software Watermarking and Fingerprinting

In order for the public to be confident in the robustness of the proposed abstract software watermarking method, it is necessary to publish the watermarking method, including the specific abstraction $\alpha$ and abstract stegosignature extractor $\overline{\mathcal{E}}$ which are used.

Therefore in practice it is more convenient to have the abstraction $\alpha_n$ and abstract extractor $\overline{\mathcal{E}}_n$ be parametrised by a secret stegokey $n$. The abstraction $\boldsymbol{\lambda} n \cdot \alpha_n$ and the abstract key extractor $\boldsymbol{\lambda} n \cdot \overline{\mathcal{E}}_n$ can be made public provided the abstract semantics $\overline{\mathcal{E}}_n(\overline{\mathcal{R}}_{\alpha_n}[\![P]\!])$ is hard, if not impossible to compute when $n$ is unknown.

Moreover, the abstract software watermarking embedders and extractors will produce different stegosignatures for different stegokeys as if we had different watermarkers, which can therefore be utilized separately and independently.

We now illustrate this principle of parametrised abstract software watermarking by a particular instance.

# 4. Abstract Software Stegosignature Embedding

## 4.1 Signatures

A secret signature will be an arbitrary large natural number $c \in \mathtt{N}$ so that $\mathcal{S} \triangleq \mathtt{N}$. In practice the natural numbers representable within the numeric primitive types must be less than or equal to a given maximum $m$ (e.g. `MAX_VALUE` for `long` in Java™). In order to bypass the implementation requirement $c \leq m$, we use the Chinese remainder theorem stating that:

*Let $n_1, \ldots, n_\ell$ be $\ell \geq 1$ positive integers which are pairwise coprime (meaning $\gcd(ni, nj) = 1$ whenever $i \neq j$) then $\mathtt{Z}/_{n_1 \ldots n_\ell \mathtt{Z}}$ is isomorphic to the cartesian product $\mathtt{Z}/_{n_1 \mathtt{Z}} \times \ldots \times \mathtt{Z}/_{n_\ell \mathtt{Z}}$.*

So in order to embed a secret signture $c$ of size strictly bounded by $n_1 \ldots n_\ell$ with $1 < n_i \leq m$ for all $i = 1, \ldots, \ell$ we can isomorphically embed $\ell$ keys $\langle c_1, \ldots, c_\ell \rangle \in [0, n_1 - 1] \times \ldots \times [0, n_\ell - 1]$ such that:

$$c = \sum_{i=1}^{\ell} \Big( \Big( \prod_{j=1}^{i-1} n_j \Big) . c_i . \Big( \prod_{j=i+1}^{\ell} n_j \Big) \Big) \pmod{\prod_{j=1}^{\ell} n_j} . \quad (6)$$

So the secret key can be assumed to be a tuple $\langle c_1, \ldots, c_\ell \rangle \in [0, n_1 - 1] \times \ldots \times [0, n_\ell - 1]$ to be embedded/extracted in a program by successively embedding/extracting $c_i, i = 1, \ldots, \ell$.

The principle of stegosignature embedding is to add a stegomark to the program in order to hide the constant $c_i$ while the extraction consists in a static analysis of the program revealing this constant $c_i$. Not knowing the secret factor $n_i$ it is computationally hard if not impossible to retrieve $c_i$. However knowing the $\ell$-ary stegokey $\langle n_1, \ldots, n_\ell \rangle$, the static analysis can be repeated $\ell$ times to retrieve the secret signature $c$. After explaining signature embedding in Sec. 4.3 and signature extraction in Sec. 5.1, we will show that the above restriction to a maximal value $m$ can be lifted.

## 4.2 Stegomark

We let $c \in \mathtt{Z}$ be a stegosignature satisfying (6) to be embedded in a program (indeed a Java™ method). As explained in Sec. 4.1, we successively embed $c_i, i = 1, \ldots, \ell$. Each $c_i, i = 1, \ldots, \ell$ is hidden in a stegomark $\mathcal{M}_r(c_i)$. The stegomark is built up in three parts, the stegomark declaration part (introducing a new auxiliary stegovariable, say `W`, hiding the value of the secret key $c_i$.):

```
int W;
```

the stegomark initialization part:

$$\mathtt{W} \;=\; P(1) \quad \text{in} \quad \mathtt{Z}$$

and the stegomark iteration part:

$$\begin{array}{rrcll} & \mathtt{W} & = & Q(\mathtt{W}) & \text{in } \mathtt{Z} \\ \text{such that:} \quad P(1) & & = & c_i & \text{in } \mathtt{Z}/_{n_i \mathtt{Z}} \quad \text{and} \\ c_i & & = & Q(c_i) & \text{in } \mathtt{Z}/_{n_i \mathtt{Z}} \end{array}$$

that is once initialized, the stegovariable `W` is constant in $\mathtt{Z}/_{n_i \mathtt{Z}}$. This property will be used to extract the stegosignature $c_i$. Although it is constant in $\mathtt{Z}/_{n_i \mathtt{Z}}$ the value of `W` will appear to be stochastic in successive executions of the stegomark iteration part $\mathtt{W} = Q(\mathtt{W})$.

## 4.3 Stegomark Inlaying

The watermarking $\mathcal{W}[\![P]\!](c_i)$ of the program $P$ consists in first choosing a method $M$, and in watermarking $M$. Formally $\mathcal{W}[\![P]\!](c_i)$ is $P$ where the chosen method $M$ is replaced by $\mathcal{W}[\![M]\!](c_i)$. In Java™, the method seems to be the smallest unit of algorithmic interest, hence our choice. It is trivial to generalize the approach to the case where the stegosignature is spread over a class or the full program.

The watermarking $\mathcal{W}[\![M]\!](c_i)$ of a method $M$ consists in inlaying the stegomark $\mathcal{M}_r(c_i)$ in the method $M$ with $\mathcal{I}[\![M, \mathcal{M}_r(c_i)]\!]$ defined as follows:

- the stegomark declaration part is included among the local declarations of the method $M$;
- the stegomark initialization part is embedded at a random position of the initial basic block of the method $M$;
- the stegomark iteration part is inserted later at a random position in the body of the method, preferably within a loop.

## 4.4 Obfuscating the Stegomark

The stegomark should be obfuscated in order to hide the stegosignature $c_i$.

First, the auxiliary variable `W` can be any integer program variable which is dead at the initialization point, in which case the extra declaration of `W` is not necessary.

The initialization assignment $\mathtt{W} = P(1)$ uses a (e.g. second-degree) polynomial $P$ of the form:

$$P(x) \;=\; x^2 + k_1 . x + k_0$$

with the coefficients $k_1 = -(1 + c_i)$ and $k_0 = 2.c_i$. In order to hide the secret signature $c_i$, these coefficients are incremented, within the limit $m$, by a random number of times the modulo $n_i$. Therefore:

$$\begin{aligned} k_1 &= -(1 + c_i) + r_1 . n_i \\ k_2 &= c_i + r_2 . n_i \end{aligned}$$

where integers $r_1$ and $r_2$ are random integers. Observe that in $\mathtt{Z}/_{n_i \mathtt{Z}}$, we have $P(1) = c_i$. Indeed, the polynomial value $P(1)$ is better computed by Horner's method:

$$P(x) \;=\; (x + k_1) . x + k_0$$

by successive assignments (where `T` is another new temporary variable[2]):

```
W = 1; ... T = W + k1; ... T = W * T; ... W = T + k0
```

introduced at random successive points in the method's initial basic block. A polynomial of higher degree can also be used if necessary.

In the iteration assignment $\mathtt{W} = Q(\mathtt{W})$ of the stegomark, $Q$ is chosen to be of the second degree (or higher), of the form:

$$Q(x) \;=\; a . x^2 + b . x + c$$

where $a$ and $b$ are (not too large) non-zero random numbers and $c$ is chosen to ensure that $c_i = Q(c_i)$:

$$c \;=\; c_i - (b . c_i + a . {c_i}^2)$$

[2] `T` can also be another program variable which is dead at that point.

Again, we use Horner's method:

$$(x) \;=\; (a.x+b).x+c$$

to compute $Q(\mathtt{W})$, by introducing the following sequence of assignments at random positions in the body of the method:

$$\mathtt{T} = \mathtt{W} * a;\ \ldots\ \mathtt{T} = \mathtt{T} + b;\ \ldots\ \mathtt{T} = \mathtt{T} * \mathtt{W};\ \ldots\ \mathtt{W} = \mathtt{T} + c\ .$$

The last instruction is inserted only if $c \neq 0$. If $c$ is negative, we use the difference $-$ instead of addition $+$ for naturalness.

# 5. Abstract Stegosignature Extraction

By defining $\alpha_i$ such that $\overline{\mathcal{R}}_{\alpha_i}[\![P]\!]$ performs constant propagation in $\mathbb{Z}/_{n_i\mathbb{Z}}$ for all methods $M$ of $P$, a static analysis will be able to recognize that some variable (i.e. $\mathtt{W}$) has the constant value $c_i$ in $\mathbb{Z}/_{n_i\mathbb{Z}}$ once initialized (or ultimately if a dead variable has been reused) thus indicating that the method $M$ had been watermarked. We now explain the technical details of this static analysis.

## 5.1 Abstract Domain and Operations

Let $n$ be some $n_1, \ldots, n_\ell$ (or their product $n_1 \ldots n_\ell$ in case $n_1 \ldots n_\ell \leq m$). We now introduce a program static analysis, parametrised by $n$ and to be repeated $\ell$ times for $n = n_1, \ldots, n_\ell$, in order to retrieve the secret signature $c$ by successively discovering $c_1, \ldots, c_\ell$.

***Basic Abstract Domains:*** We let $\mathbb{Z}/_{n\mathbb{Z}} \triangleq \{\{hz \mid h \in \mathbb{Z}\} \mid 0 \leq z < n\} \cong [0, n-1]$ be the ring of integers modulo $n > 1$. The unit element for addition is 0 and that for multiplication is 1.

The basic abstract domain is that of constant propagation [13] for the ring $\mathbb{Z}/_{n\mathbb{Z}}$ that is the complete lattice:

$$L_n \quad \triangleq \quad \{\bot, \top\} \cup \mathbb{Z}/_{n\mathbb{Z}}$$

where $\bot$ abstracts the empty set (that is unreachable code), $\top$ abstracts $\mathbb{Z}/_{n\mathbb{Z}}$ (that is an unknown value) and $c \in \mathbb{Z}/_{n\mathbb{Z}}$ abstracts the congruence class $\{hc \mid h \in \mathbb{Z}\}$. The set $L_n$ is ordered by:

$$\forall x \in \mathbb{Z}/_{n\mathbb{Z}} : \bot \sqsubseteq \bot \;@\; x \sqsubseteq x \;@\; \top \sqsubseteq \top$$

which abstracts set inclusion. $L_n$ is a complete lattice where the join $\sqcup$ (abstracting set union $\cup$) and the meet $\sqcap$ (abstracting set intersection $\cap$) are defined, for all $x \in L_n$, $y \in \mathbb{Z}/_{n\mathbb{Z}}$ such that $x \neq y$, by:

$$\begin{array}{lllll} \bot \sqcup x & = & x \sqcup \bot & = & x \\ \top \sqcup x & = & x \sqcup \top & = & \top \\ \bot \sqcap x & = & x \sqcap \bot & = & \bot \\ \top \sqcap x & = & x \sqcap \top & = & x \end{array} \qquad \begin{array}{lll} x \sqcup x & = & x \\ x \sqcup y & = & \top \\ x \sqcap x & = & x \\ x \sqcap y & = & \bot\ . \end{array}$$

***Basic Abstractions:*** We let $\eta_n \in \mathbb{Z} \mapsto \mathbb{Z}/_{n\mathbb{Z}}$ defined as:

$$\eta_n(z) \triangleq \text{let} \quad r = z \bmod n \quad \text{in} \quad \text{if } r \geq 0 \text{ then } r \text{ else } r+n$$

be the canonical embedding of the set $\mathbb{Z}$ of integers onto $[0, n-1]$ isomorphic to the quotient ring $\mathbb{Z}/_{n\mathbb{Z}}$. The abstraction $\alpha_n$ specifies how a subset of $\mathbb{Z}/_{n\mathbb{Z}}$ should be approximated. The intuition is that the empty set ($\emptyset$ represented by $\bot$) and equivalence classes ($\{hc \mid h \in \mathbb{Z}\}$ represented by $\eta_n(c)$ where $c \in \mathbb{Z}$) are preserved exactly while any other subset is approximated by $\mathbb{Z}/_{n\mathbb{Z}}$ (represented by $\top$). Formally, $\alpha_n$ is defined by:

$$\alpha_n(Z) \triangleq \bigsqcup \{\eta_n(z) \mid z \in Z\}\ .$$

The concretization $\gamma_n$ specifies which subsets of $\mathbb{Z}/_{n\mathbb{Z}}$ are represented by elements of the abstract domain $L_n$:

$$\begin{array}{rcl} \gamma_n(\bot) & \triangleq & \emptyset, \\ \gamma_n(z) & \triangleq & \{hz \mid h \in \mathbb{Z}\} \quad \text{if} \quad z \in [0, n-1], \\ \gamma_n(\top) & \triangleq & \mathbb{Z}\ . \end{array}$$

We obtain a Galois connection [7]:

$$\langle \wp(\mathbb{Z}), \subseteq \rangle \underset{\alpha_n}{\overset{\gamma_n}{\leftrightarrows}} \langle L_n, \sqsubseteq \rangle$$

such that for all $Z \in \wp(\mathbb{Z})$ and $\overline{Z} \in L_n$:

$$(\alpha_n(Z) \sqsubseteq \overline{Z}) \iff (Z \subseteq \gamma_n(\overline{Z}))\ .$$

***Basic Abstract Operations:*** The abstraction $\odot$ of a $k$-ary concrete operation $\cdot$ (with domain $\mathrm{dom}(\cdot)$ and codomain $\mathrm{codom}(\cdot)$) should satisfy for all $\overline{x}_1 \in L_n, \ldots, \overline{x}_k \in L_n$ [7]:

$$\begin{array}{l} \{\cdot(x_1, \ldots, x_k) \mid x_1 \in \gamma_n(\overline{x}_1) \wedge \ldots \wedge x_k \in \gamma_n(\overline{x}_k) \wedge \\ \qquad\qquad (x_1, \ldots, x_k) \in \mathrm{dom}(\cdot)\} \quad \subseteq \quad \gamma_n(\odot(\overline{x}_1, \ldots, \overline{x}_k))\ . \end{array}$$

The abstract unary ($k = 1$) operation $\odot$ (such as the abstract inverse $\ominus$) abstracting the concrete operation $\cdot$ (of the inverse $-$) is defined as follows:

$$\begin{array}{rcll} \odot\bot & \triangleq & \bot, \qquad \odot\top \triangleq \top, & \\ \odot z & \triangleq & \alpha_n(\{\cdot x \mid x \in \gamma_n(z) \cap \mathrm{dom}(\cdot)\}) & \text{if} \quad z \in [0, n-1] \\ & = & \eta_n(\cdot z)\ . & \end{array}$$

The abstract binary ($k = 2$) operations $\odot$ (of addition $\oplus$, subtraction $\ominus$ or multiplication $\otimes$) for the concrete operation $\cdot$ (of addition $+$, subtraction $-$ or multiplication $\times$) are defined as follows:

$$\begin{array}{rcll} \bot \odot z & \triangleq & \bot, \qquad \odot\bot \triangleq \bot, & \text{if} \quad z \in L_n, \\ z_1 \odot z_2 & \triangleq & \alpha_n(\ \{x_1 \cdot x_2 \mid x_1 \in \gamma_n(z_1) \wedge x_2 \in \gamma_n(z_2) \wedge & \\ & & \qquad (x_1, x_2) \in \mathrm{dom}(\cdot)\}) & \text{if} \quad z_1, z_2 \in [0, n-1] \\ & = & \eta_n(z_1 \cdot z_2), & \\ \top \odot z & \triangleq & \top, \qquad z \odot \top \triangleq \top & \text{if} \quad z \in L_n\ . \end{array}$$

***Abstract Integer Variable Domain:*** Let $n_1, \ldots, n_\ell$ be $\ell \geq 1$ positive integers which are which are pairwise coprime. The abstract domain is the product

$$\langle L, \dot{\sqsubseteq} \rangle \quad \triangleq \quad \langle \prod_{i=1}^{\ell} L_{n_i}, \dot{\sqsubseteq} \rangle \tag{7}$$

smashed for $\bot$ and $\top$ which, by the Chinese remainder theorem, is isomorphic to the abstract domain:

$$\langle L, \sqsubseteq \rangle \quad \triangleq \quad \langle \{\bot, \top\} \cup \mathbb{Z}/_{n_1 \ldots n_\ell \mathbb{Z}}, \sqsubseteq \rangle\ . \tag{8}$$

When $n_1 \ldots n_\ell > m$ the analysis on (8) cannot be directly implemented using machine integers. It can be replaced by an analysis with abstract domain (7), which is always more precise (and even strictly more precise since one can discover that a variable is constant modulo some $n_j$ although it might not be constant modulo the product $n_1 \ldots n_\ell$). Moreover a static analysis with (7) is equivalent to $\ell$ successive analyzes with $L_n = L_{n_1}, \ldots, L_n = L_{n_\ell}$.

***Abstract Environment Domain:*** Given the set $\mathtt{X}[\![P]\!]$ of variables of program $P$, the abstract domain is extended pointwise to environments mapping variables to integer values. Recall that the set of memory states is $\mathcal{M}[\![P]\!] \triangleq \mathtt{X}[\![P]\!] \mapsto \mathcal{V}[\![P]\!]$. We let

$\mathtt{X}_{\mathtt{int}}[\![P]\!] \subseteq \mathtt{X}[\![P]\!]$ be the subset of program variables which are of integer type. We have:

$$\langle \wp(\mathcal{M}[\![P]\!]), \dot{\subseteq} \rangle \underset{\dot{\alpha}_{n_1 \ldots n_\ell}}{\overset{\dot{\gamma}_{n_1 \ldots n_\ell}}{\leftrightarrows}} \langle \overline{\mathcal{M}}[\![P]\!], \ddot{\sqsubseteq} \rangle$$

where $$\overline{\mathcal{M}}[\![P]\!] \stackrel{\Delta}{=} \mathtt{X}_{\mathtt{int}}[\![P]\!] \mapsto \prod_{i=1}^{\ell} L_{n_i}$$

and for all $m \subseteq \mathcal{M}[\![P]\!]$ and $\overline{m} \in \overline{\mathcal{M}}[\![P]\!]$:

$$\dot{\alpha}_{n_1 \ldots n_\ell}(m) \stackrel{\Delta}{=} \boldsymbol{\lambda}\, \mathtt{X} \in \mathtt{X}_{\mathtt{int}}[\![P]\!] \cdot \prod_{i=1}^{\ell} \alpha_{n_i}(\{\rho(\mathtt{X}) \mid \rho \in m\})$$

$$\dot{\gamma}_{n_1 \ldots n_\ell}(\overline{m}) \stackrel{\Delta}{=} \{\rho \in \mathcal{M}[\![P]\!] \mid \forall \mathtt{X} \in \mathtt{X}_{\mathtt{int}}[\![P]\!] : \forall i \in [1, \ell] : \rho(\mathtt{X}) \in m_{n_i}(\overline{m}(\mathtt{X})_i)\} .$$

All information on non-integer variables is lost while the information on integer variables is restricted to modular constant information.

***Abstract Reachability Domain:*** Given the set $\mathcal{C}[\![P]\!]$ of control points $c$ of a program $P$, the abstract environment domain is extended pointwise to all program points (recall that $\Sigma[\![P]\!] = \mathcal{C}[\![P]\!] \times \mathcal{M}[\![P]\!]$). The abstract domain $L[\![P]\!]$ which is used for signature extraction in a program $P$ is therefore:

$$L[\![P]\!] \stackrel{\Delta}{=} \mathcal{C}[\![P]\!] \mapsto \overline{\mathcal{M}}[\![P]\!]$$

satisfies the requirement (1) since:

$$\langle \wp(\Sigma[\![P]\!]), \subseteq \rangle \underset{\ddot{\alpha}_{n_1 \ldots n_\ell}}{\overset{\ddot{\gamma}_{n_1 \ldots n_\ell}}{\leftrightarrows}} \langle L[\![P]\!], \dot{\sqsubseteq} \rangle \tag{9}$$

where given $S \subseteq \Sigma[\![P]\!]$ and $\overline{S} \in L[\![P]\!]$:

$$\ddot{\alpha}_{n_1 \ldots n_\ell}(S) \stackrel{\Delta}{=} \prod_{c \in \mathcal{C}[\![P]\!]} \dot{\alpha}_{n_1 \ldots n_\ell}(\{m \mid \langle c, m \rangle \in S\}),$$

$$\ddot{\gamma}_{n_1 \ldots n_\ell}(\overline{S}) \stackrel{\Delta}{=} \{\langle c, m \rangle \mid c \in \mathcal{C}[\![P]\!] \wedge m \in \overline{S}(c)\} .$$

***Static Analysis:***

- ***Collecting Semantics:*** The considered concrete collecting semantics $\mathcal{R}[\![P]\!]$ of each method $P$ is the set of descendants of the initial states of the method $P$ with an initial environment $\boldsymbol{\lambda}\, \mathtt{X} \cdot .\mathcal{V}[\![P]\!]$ where all variables are undefined.

- ***Abstraction:*** The extraction of the secret signature for the stegokey $\langle n_1, \ldots, n_\ell \rangle$ starts with the computation of an overapproximation of the abstract semantics $\overline{\mathcal{R}}_{\ddot{\alpha}_{n_1 \ldots n_\ell}}[\![P]\!]$ that is by constant propagation in the abstract domain $L[\![P]\!]$ for each method $P$ of the program. Therefore the static analysis is purely intraprocedural on local variables of integer type using the abstraction (9). In practice, this is equivalent to the successive (or simultaneous or parallel) static analyses propagating constants in the abstract domains corresponding to the individual $L_{n_i}$, $i = 1, \ldots, \ell$ since:

$$\overline{\mathcal{R}}_{\ddot{\alpha}_{n_1 \ldots n_\ell}}[\![P]\!] = \prod_{i=1}^{\ell} \overline{\mathcal{R}}_{\ddot{\alpha}_{n_i}}[\![P]\!] .$$

- ***Abstract Transformer:*** To simplify the abstract transformer $\overline{\mathcal{F}}_{\ddot{\alpha}_{n_1 \ldots n_\ell}}[\![P]\!]$ is defined using the following approximations (further refinements to deter obfuscation may be necessary as discussed later):

  - the initial state of the integer local variables is undefined ($\top$);
  - the only operations taken into account in the control flow graph are the addition, the subtraction and the multiplication;
  - tests as well as other operations such as procedure and function calls are simply ignored;
  - for loops, tests and branching the environment pointwise union is used at control junction points.

- ***Abstract Semantics:*** The abstract semantics $\overline{\mathcal{R}}_{\ddot{\alpha}_{n_1 \ldots n_\ell}}[\![P]\!]$ of the method $P$ is defined as the least fixpoint of the abstract transformer $\overline{\mathcal{F}}_{\ddot{\alpha}_{n_1 \ldots n_\ell}}[\![P]\!]$ as defined in (2). It can be computed iteratively (3) using any *chaotic* or *asynchronous* iteration strategy, as usual in abstract interpretation.

The result of the analysis is the set of constants belonging to the ring $\mathbb{Z}/_{n_i \mathbb{Z}}$ which are the constant values of local integer variables of the method $P$ for $i = 1, \ldots, \ell$.

***Application of Static Analysis to Validation Watermarking:*** Validation watermarking [15] consists in embedding a watermark in a software, which may be visible hence simply concatenated to the software, and yields the essence of the software. This digest can be used to verify that the software is still essentially the same as when authored. The essence extraction process can be (a cryptographic digest of) the abstract reachability analysis $\overline{\mathcal{R}}_{\alpha}[\![P]\!]$ for a secret abstraction $\alpha$ chosen such that it is essentially invariant for all versions of the subject software. Then a publicly available verifier can detach the watermark from the software, make the static analysis of the software and compare the result with the watermark to check that the software (hence, its abstract semantics) have not been modified. Observe that the verifier itself must not be faked whence its integrity should be ensured by a signature or a MD5 checksum.

***Signature Extraction:*** The extractor (5) uses the result $\overline{\mathcal{R}}_{\ddot{\alpha}_{n_1 \ldots n_\ell}}[\![P]\!]$ of the static analysis of the method $P$ to extract the stegosignature.

The stegosignature $c$ is extracted by the abstract extractor $\overline{\mathcal{E}}$ if and only if all $c_i$, $i = 1, \ldots, \ell$ are extracted from the abstract semantics $\overline{\mathcal{R}}_{\ddot{\alpha}_{n_1 \ldots n_\ell}}[\![P]\!]$ of the method $P$.

To extract $c_i$, the abstract extractor $\overline{\mathcal{E}}$ has to determine whether the static analysis has detected that some local integer variable of the method $P$ has the abstract value $c_i$ at two program points at least (at least one should appear after the stegomark initialization part and another one the stegomark iteration part).

Despite the overapproximation (4), signature extraction from the watermarked method always succeeds:

THEOREM 1 (CORRECTNESS). *For all methods $P \in \mathcal{P}$, stegokey $n_1 \ldots n_\ell$, stegosignature $c = c_1 \ldots c_\ell$, the abstract extractor $\overline{\mathcal{E}}_{n_1 \ldots n_\ell}$ will extract the stegosignature $c$ from the watermarked method $\mathcal{W}_{n_1 \ldots n_\ell}[\![P]\!](c)$ that is $c \in \overline{\mathcal{E}}_{n_1 \ldots n_\ell}(\overline{\mathcal{R}}_{\ddot{\alpha}_{n_1 \ldots n_\ell}}[\![\mathcal{W}_{n_1 \ldots n_\ell}[\![P]\!](c)]\!])$.*

More generally, we would like to prevent attacks by program transformation. Ideally, the probability that there exist a polynomial syntactic program transformation algorithm $T$ that preserves the semantics $S[\![P]\!]$ of program $P$ (may be up to some observational abstraction $\alpha_{\mathcal{O}}$: $\forall P \in \mathcal{P} : \alpha_{\mathcal{O}}(S[\![P]\!]) = \alpha_{\mathcal{O}}(S[\![T[\![P]\!]]\!])$) and that can attack the watermarked program (i.e. $c \notin \overline{\mathcal{E}}_{n_1 \ldots n_\ell}(\overline{\mathcal{R}}_{\ddot{\alpha}_{n_1 \ldots n_\ell}}[\![T[\![\mathcal{W}_{n_1 \ldots n_\ell}[\![P]\!](c)]\!]]\!])$) should be very low.

The characterization of all such transformations $T$ is quite difficult. Hence such an ideal theorem would be hard to prove. Potential attacks are further discussed in Sec. 7, 8 and 9.

## 5.2 Example

***Subject Program:*** Let us consider the embedding of the stegosignature in the `main` method of the following sample program:

```
public class Fibonacci {
  public Fibonacci() {}
  public static void main(String[] args){
    int n=Integer.parseInt(args[0]);
    int a=0; int b=1;
    for (int i=1;i<n;i++)
      { int c=a+b; a=b; b=c; }
    System.out.println("Fib("+n+") = "+b);}}
```

For this program, we are only interested in the values of `n` and `b` at the final states reachable from the initial states as defined by its operational semantics (and this defines its observational abstraction $\alpha_{\mathcal{O}}$).

***Stegokey and Stegosignature:*** The secret stegokey is assumed to be $\ell = 2$, $n_1 = 30001$ et $n_2 = 5421$. The stegosignature is $c_1 = 21349$ and $c_2 = 3012$.

***Stegomark:*** The stegoprogram is obtained by inclusion of the following stegomark (where, for clarity, the variables `W` and `T` are given explicit names of the form <W:$n_i$:$c_i$> and <T:$n_i$:$c_i$>, $1 \leq i \leq \ell$ which, for discretion, should be chosen as more anonymous identifiers by obfuscation).

- ***Stegomark for $c_1$:***

```
int <W:30001:21349>, <T:30001:21349>;

<W:30001:21349> = 1;
<T:30001:21349> = <W:30001:21349> - 111353;
<T:30001:21349> = <W:30001:21349> * <T:30001:21349>;
<W:30001:21349> = <T:30001:21349> - 47305;

<T:30001:21349> = <W:30001:21349> * 4;
<T:30001:21349> = <T:30001:21349> + 1566;
<T:30001:21349> = <T:30001:21349> * <W:30001:21349>;
<W:30001:21349> = <T:30001:21349> + 21494;
```

- ***Stegomark for $c_2$:***

```
int <W:5421:3012>, <T:5421:3012>;

<W:5421:3012> = 1;
<T:5421:3012>=<W:5421:3012>+-35539;
<T:5421:3012>=<W:5421:3012>*<T:5421:3012>;
<W:5421:3012>=<T:5421:3012>+11445;

<T:5421:3012>=<W:5421:3012>*658;
<T:5421:3012>=<T:5421:3012>+971;
<T:5421:3012>=<T:5421:3012>*<W:5421:3012>;
<W:5421:3012>=<T:5421:3012>+4623;
```

- ***Stegoprogram:***

```
public class FibonacciWatermarked {
  public FibonacciWatermarked() {}
  public static void main(String[] args){
    int n=Integer.parseInt(args[0]);
    int a=0; int b=1;  int d=1; int e=35538; int f=1;
    int g=-111353;
    e=d*e; d=e+11445; g=f*g; f=g-47305;
    for (int i=1;i<n;i++)
    { int c=a+b; e=d*658; f=f*4; a=b; g=g+1566; e=e+971;
      g=g*f; e=e*d; b=c; d=e+4623; f=g+21494; }
    System.out.println("Fib("+n+") = "+b); }}
```

## 5.3 Lifting the Data Size Physical Limitation

The physical data size limitation, as given by the maximal integer $m$ which can be represented by the type `int` in our sample abstract watermarking algorithm, can be lifted by considering integers of arbitrary size in the stegomark and stegoprogram non-standard semantics whence in the abstract semantics $\overline{\mathcal{R}}_{\ddot{\alpha}_{n_1 \ldots n_\ell}}[\![P]\!]$ computed by the static analyser.

For the static analyzer, the concrete interpretation of modulo arithmetic operations is in `Z` (whence non-standard for the program semantics). To handle this non-standard semantics correctly, the static analyzer must use libraries of big numbers. Of course such big numbers could also have been used in the stegomark but this would be too easily recognizable.

The concrete execution of the instructions of the stegomark inlaid within the stegoprogram may overflow. Fortunately this is completely harmless with the standard integer modulo arithmetic. The only limitation to be taken into account is for the constants (like $k_0$, $k_1$, $a$, $b$, $c$) appearing in the text of the stegomark, which must be within the physical limitation imposed by the language. When too large in the standard concrete semantics, these constants can be computed in the non-standard semantics by program expressions using only constants satisfying the physical limitation imposed by the language standard semantics. Actual evaluation of the stegomark (with overflow in the standard concrete semantics but not in the non-standard one) will then be harmless thanks to modulo arithmetic.

In case $m$ is chosen much larger than the physical limitation and is kept secret, the stegosignature extraction is even harder if not impossible.

# 6. Requirements Satisfied by Abstract Software Watermarking

Our abstract software watermarking method satisfies a number of criteria discussed below which are advisable for all software watermarking methods (e.g. to be effective in the protection of the ownership of intellectual property).

## 6.1 Practical Requirements Satisfied by the Watermarks

***Functionality Preservation:*** The watermarking should preserve the functionality of the subject program and so the semantics of the stegoprogram $\mathcal{W}[\![P]\!](s)$ should be "identical" as that of the subject program $P$ [3]. Formally, this means that up to some observational abstraction $\alpha_{\mathcal{O}}$, the operational/denotational semantics `S` are the same and so for all $P \in \mathcal{P}$ and $s \in \mathcal{S}$:

$$\alpha_{\mathcal{O}}(\mathtt{S}\,[\![P]\!]) = \alpha_{\mathcal{O}}(\mathtt{S}\,[\![\mathcal{W}[\![P]\!](s)]\!]) = \alpha_{\mathcal{O}}(\mathtt{S}\,[\![\mathcal{I}[\![P, \mathcal{M}_r(s)]\!]]\!])\,.$$

Typically the abstraction $\alpha_{\mathcal{O}}$ gets rid of the auxiliary variables and the effect of the code which are inlaid in the subject program to encode the signature.

***Performance Preservation:*** The performance of the subject program should not be significantly degraded. Our abstract software watermarking method preserves execution time up to some small constant factor (which is negligible for large programs).

[3] up to e.g. a little more time and memory consumption for execution of the stegomark.

***Universality Preservation:*** If the subject program is universal (i.e. can be executed on any hardware with appropriate compiler or interpreter) then no special hardware should be required for executing the stegoprogram (contrary to native code with encrypted signatures [6]).

***Unbounded Signature Size*** There should be no bound on the size of the signature (or it should be very large) thus allowing embedded signatures to be arbitrarily encrypted unique identifiers. This is achieved both by decomposition of signatures $c$ of size strictly bounded by $n_1 \ldots n_\ell$ into $\ell$ keys $\langle c_1, \ldots, c_\ell \rangle \in [0, n_1 - 1] \times \ldots \times [0, n_\ell - 1]$ [Sec. 4.1] and by choosing large $n_i$, $i = 1, \ldots, n$ beyond the machine limitation $m$ thanks to a non-standard reinterpretation of modulo arithmetic in $\mathtt{Z}$ [Sec. 5.3].

***Credibility and False Recognition:*** Since the signature uniquely identifies the copyright owner, the watermarking should provide an authentic, clear, secure and indubitable proof that the stegoprogram is protected (as opposed to a false recognition or a probabilistic detection). For the watermarking to be credible, most programs should be unmarked that is the extraction of stegosignature from programs in which no signature has been embedded should not produce a false recognition. Formally, for all $P$, $Q \in \mathcal{P}$ and $s \in \mathcal{S}$ such that $Q \neq \mathcal{W}[\![P]\!](s)$, we should have:

$$s \notin \mathcal{E}[\![Q]\!] \,. \tag{10}$$

Of course, theorem 5.1 does not exclude false positives (since we want to be able to watermark the same method several times with the same stegokey). This means that it is possible to find a subject program variable which happens to be a constant $\kappa$ in $L_{n_i}$ for some $i \in [1, \ell]$. This constant $\kappa$ might create an ambiguous result at extraction time whenever $0 \neq \kappa \neq c_i$ in contradiction with (10).

A simple solution is to perform a signature extraction just after signature embedding to check that this does not happen. If this happens one can either change the stegokey $n_i$ used for embedding/extraction, or change the stegosignature $c_i$ in the secret database of stegosignatures (for the given stegokey $n_i$), or maintain $\kappa$ as invalid for the stegosignature $c_i$ for the program $P$ in the secret database for the stegokey $n_i$, etc.

***Secrecy:*** The watermark, i.e. the stegomark inlaid in the subject program, should not reveal the signature when discovered by the average observer (but should be readily detectable by the proper authorities). In case the stegomark can reveal the signature, it is a good practice to encrypt the signature encoded within the stegomark.

• ***Extracting the Signature from the Stegomark:*** It should be impossible, or at least computationally hard, to extract the stegosignature $s \in \mathcal{S}$ from the stegomark $\mathcal{M}_r(s)$. As in many cryptographic methods, this is based on the use of a random stegokey $n \in \mathtt{N}$ which is kept secret, so that the stegomark:

$$\mathtt{int\ W;\ \ldots\ W} = P(1);\ \ldots\ \mathtt{W} = Q(\mathtt{W})$$

can hardly reveal the signature which is left invariant by the stegomark computation. Indeed, given polynomials $P$ and $Q$, the question is to solve for the unknown $x, n$ where:

$$\begin{aligned} x &= P(1) \bmod n \\ x &= Q(x) \bmod n \,. \end{aligned}$$

Following [12], let us set $Q'(X) = Q(X) - X$, so we now have

$$\begin{aligned} x &= P(1) \bmod n \\ 0 &= Q'(x) \bmod n \\ \text{or equivalently} \quad 0 &= Q'(P(1)) \bmod n. \end{aligned}$$

Anyone can compute $Q'(P(1))$, which is some number $i$, and the problem is now to find $n$ such that $i = 0 \bmod n$ which essentially amounts to factor $i$ in order to find one factor of $i$ (or even all of them). Factoring can reasonably be assumed to be hard for large factors. Hence one might want to check that $i$ has large factors (which is easier when already knowing the factor $n$) and consider lifting the data size physical limitation as explained above. In practice extracting the stegosignature $s$ without knowing the secret stegokey $n$ then essentially amounts to randomly trying all possible stegokeys $n \in \mathtt{N}$ (or, at least a very large number of the possible stegokeys).

• ***Extracting the Stegomark from the Stegoprogram:*** Finally, it should be impossible, or at least computationally hard, to automatically discover the stegomark within the watermarked program $\mathcal{W}[\![P]\!](s) = \mathcal{I}[\![P, \mathcal{M}_r(s)]\!]$. Obfuscation methods are helpful for that purpose. Again the static analysis method for signature extraction proposed in Sec. 5.1 $\mathcal{E}_n[\![\mathcal{W}_n[\![P]\!](s)]\!]$ depends on the secret stegokey $n$ so that when $n$ is unknown, the signature extraction essentially amounts to randomly trying all possible stegokeys.

***Robustness/Perenniality:*** The watermarks should be permanent. If visible, they should be hard or impossible to remove without investing a lot of time and/or without severely damaging the stegoprogram so much that it becomes hardly usefully usable or leave traces on the modified stegoprogram which can be immediately detected by comparison with the undamaged subject program.

If not impossible, it should be computationally hard to recover the subject program $P$ from the stegoprogram $\mathcal{W}[\![P]\!](s) = \mathcal{I}[\![P, \mathcal{M}_r(s)]\!]$. One solution is for the software watermarker $\mathcal{W}[\![P]\!](s)$ to include an obfuscation of the stegoprogram. However this is not mandatory since obfuscation pursues different objectives. Moreover, and contrary to [2], we do not aim at obfuscating the observable semantics $\alpha_\mathcal{O}(\mathtt{S}\ [\![P]\!])$ of the program $P$ (which can be specified e.g. in a publicly available reference manual).

***Multiple Watermarks:*** Abstract software watermarking allows several different signatures to be embedded in the stegoprogram hence is robust against rewatermarking. Formally, marking with a new signature should not delete previous signatures:

$$\text{if} \quad s \in \mathcal{E}[\![P]\!] \quad \text{then} \quad s \in \mathcal{E}[\![\mathcal{W}[\![P]\!](s')]\!] \,.$$

The number of signatures should be unbounded (or very large) at the time that the subject program is created. Indeed the value of the stegokey $n$ fixes the number of possible stegosignatures for that stegokeys, but the number of stegokeys is itself unbounded. Moreover if a program is signed several times, the extractors are only able to recover the signatures for which they are authorized provided they are given different stegokeys $n$.

***Fingerprinting:*** The abstract software watermarking method does allow fingerprinting to uniquely mark each program for every buyer by a unique licence number. If that buyer then makes an illicit copy, the illicit duplication may be convincingly demonstrated by extracting the stegosignature which is the given licence number.

## 6.2 Practical Requirements Satisfied by the Abstract Embedding/Extraction Algorithms

Our stegosignature embedding/extraction algorithms satisfy the following requirements which are desirable for all software watermarking tools.

***Automaticity:*** Signature embedding and extraction are fully automatic and require no manual preparation of the subject program. Hence they are usable on a large scale and allow for checking of legal use, e.g. on the web.

***Low Cost:*** Signature embedding and extraction have a low computational complexity, comparable to compilation.

***Resistance to Counterfeiting:*** The watermark should withstand direct and automatic attacks (e.g. by creation of counterfeit of the subject program using typical automatic program transformations that are common to program manipulation applications such as obfuscation) but not disallow the copying of the signed file.

Formally, the watermarking should resist program transformations $T \in \mathcal{P} \mapsto \mathcal{P}$ that do not change the observable abstraction of the subject program semantics. This means that for all $P \in \mathcal{P}$, $s \in \mathcal{S}$:

$$\text{if} \quad \alpha_{\mathcal{O}}(\mathtt{S}\ [\![\mathcal{W}[\![P]\!](s)]\!]) = \alpha_{\mathcal{O}}(\mathtt{S}\ [\![T[\mathcal{W}[\![P]\!](s)]]\!]) \quad \text{then}$$
$$\mathcal{E}[\![\mathcal{W}[\![P]\!](s)]\!] = \mathcal{E}[\![T[\mathcal{W}[\![P]\!](s)]]\!] \ .$$

In our case, the static analysis can be made more difficult so this problem is further discussed in Sec. 9.

***Resistance to Fraudulent Reuse Including with Hardware Protection:*** Since extraction requires no execution of the stegoprograms at all, it is possible to extract the signature of the stegoprogram if only part of it is available but not executable (provided obviously that the available part of this stegoprogram contains the stegosignature). This makes possible the tracking of stegoprograms on the web or of parts of the stegoprogram included in another program itself protected e.g. by a hardware dongle.

***Public Domain:*** The embedding/extraction algorithms can be made public since they require a secret stegokey (but not the subject program) for extraction. Moreover different extractors can be authorized owning different stegokeys without possible interactions between them.

***Pervasion:*** The embedding/extraction of the signature in the stegomark is symmetric since both depend on the same secret stegokey ($n$ in Sec. 6.1). As noted by [12], it may be absolutely necessary to disclose information from time to time which may require, in absence of trusted third authority, to make public the secret stegokey. In this case, copies of the software previously watermarked using the same stegokey, will be unprotected after the first action taken to enforce the watermark.

One solution is to use fingerprinting, that is a unique stegomark for each copy of the software with different stegokeys or to insert several stegomarks in all copies.

A complementary solution is to choose the stegosignature $s$ and the stegokey $n$ as large primes (assuming again that the data size physical limitation is lifted for unbounded signature size as in Sec. 5.3). The stegoprogram is marked with $s$ by inserting the stegomark initialization part:

$$\mathtt{W} \quad = \quad P(1) \qquad \text{in} \quad \mathtt{Z}$$

and the stegomark iteration part:

$$\mathtt{W} \quad = \quad Q(\mathtt{W}) \qquad \text{in} \quad \mathtt{Z}$$

$$\text{such that:} \qquad P(1) \quad = \quad s \qquad \text{in} \quad \mathtt{Z}/_{n\mathtt{Z}} \quad \text{and}$$
$$s \quad = \quad Q(s) \quad \text{in} \quad \mathtt{Z}/_{n\mathtt{Z}}$$

as discussed in Sec. 4.2. The stegoprogram is then deposited with accompanying signatures $s.p_i$ where the $p_i$, $i = 1, \ldots, k$ are large prime numbers for safekeeping by one or better several trusted third parties. In order for an unbiased third party to equitably verify that the stegoprogram is signed, the verifier is given (at the $i$-th verification):

- the stegosignature $s.p_i$;
- the abstract stegosignature extractor $\overline{\mathcal{E}}$;
- the stegokey $n.p_i$.

$$\text{By ensuring that} \quad P(1) \quad = \quad s.p_i \qquad \text{in} \quad \mathtt{Z}/_{n.p_i\mathtt{Z}} \quad \text{and}$$
$$s.p_i \quad = \quad Q(s.p_i) \quad \text{in} \quad \mathtt{Z}/_{n.p_i\mathtt{Z}}$$

the verifier can check that the stegoprogram is signed by $s.p_i$ and can do the same with the originally signed program.

If the information $\langle \overline{\mathcal{E}}, s.p_i, n.p_i \rangle$ is made public, claimants might be able to discover the signature $s.p_i$ by designing their own program analyser. This may be a good reason to keep the extractor $\overline{\mathcal{E}}$ private, or at least to have the extractor not reveal where the stegomark is within the stegoprogram, and shows the necessity for deterring attacks on stegomarks as discussed in next Sec 7. Nevertheless, neither $s$ nor $n$ can be computationally discovered so that $\langle s.p_{i+1}, n.p_{i+1} \rangle$ can be used for the next verification of stegoprograms whose stegomark is persistent.

# 7. General Attacks on Signatures and Stegomarks

Manual attacks against watermarked programs can hardly be avoided if enough manpower and time are available, so we concentrate on automatic attacks. Let us recall the various attacks considered by [3]:

**Subtractive attacks** detect the presence and approximate location of the stegosignatures and eliminate the part of the program where it is supposed to be located. Examples of subtractive attacks include static dead code elimination in case the stegosignatures are supposed to be hidden in dead code (see e.g. [14]) or dynamic observation of the dead code in case the dead code is protected by an opaque predicate (*opaque* means that the outcome of the predicate is known at watermarking time but the predicate is difficult for an adversary to resolve i.e. to find the truth value solution of [1, 6, 14]).

**Distortive attacks** apply transformations to the object so that the stegosignatures can no longer be extracted. Obfuscation and optimizing compilation to generate machine code are such distortive attacks. Another example is [19] where secrets are hidden in sequences of machine code, which can be easily distorted by replacing machine instructions by equivalent ones.

**Additive attacks** watermark with new signatures so that one cannot be proved that the original mark temporally precedes the pirate ones.

**Collusive attacks** remove signatures by comparison of different versions of the same program watermarked by different fingerprints. An example would be `diff` (which is naïve since it can be easily defeated by obfuscation).

For example [3, 16] encodes signatures into graphs generated when executing the program for special inputs. Program monitoring or a probabilistic static analysis can be used to discover the parts of the program that are seldom executed. Then a dependence analysis as in program slicing can be used as a subtractive attack eliminating the part of the program producing the graph. A distortive attack would modify the graph whence the signature.

# 8. General Attack Deterrence

**Subtractive attacks** are made difficult if the elimination of the signature changes/destroys the semantics of the program so that it become unusable. Therefore a good strategy is to make the stegomark dependent upon the subject program and reciprocally e.g. by transforming the subject code so that some values are computed as functions of the stegomark or have original and stegovariables merged, a well-known distortive attack!

**Distortive attacks** may not all be disturbing. For example obfuscation makes reverse engineering even more difficult so might sometime be considered helpful. The same way code generation prevents easy redistribution hence is also a form of protection.

Some distortive attacks can be prevented by considering only part of the code to extract signatures. Many of the obfuscation methods considered in [5, 6] can be defeated in this way. For example introducing dead and irrelevant code or converting a reducible to an irreducible flow graph does not change the abstract interpretation of the useful code. The same way, opaque test and loop predicates [1, 6, 14] is no problem if the signature extraction does not depend upon predicates. However the embedding might include such opaque predicates for obfuscation purposes. Restructurations of classes (such as modifying inheritance relations, extending the inheritance hierarchy tree, false refactoring, method inlining, clone methods as considered in [5]) are ineffective if the signatures are embedded at the method level and the signature extraction is purely local, not depending on global variables.

Some other distortive attacks can be prevented by considering only part of the program data to extract signatures. Among the obfuscation methods, array restructuration considered in [5] and object aliasing considered in [6] can be simply defeated by putting no data in structured static or dynamic data.

**Additive attacks** are difficult to fight in particular if the embedding algorithm is made public. Note however that unique signatures as well as original and signed programs can be revealed to trustworthy authorities at the embedding time to authenticate temporal precedence whence determine the actual owner of the program.

**Collusive attacks** can be prevented by allowing the embedding of multiple signatures. A common initial watermark can be embedded in all copies. Moreover obfuscation of the copies using different program transformations (including e.g. different code reorderings) would make comparisons very difficult.

Obviously not all possible attacks have been considered. The most harmful ones will be discussed in next Sec. 9.

# 9. Possible Harmful Specific Attacks

The considered signature embedding and extraction methods disarm the general attacks considered in Sec. 7. We now consider specific attacks against the protocol as described in [3, 6] that might be harmful. Obfuscation methods are obvious candidates for preventing signature extraction by making static analysis difficult, if not impossible. First note that if the protection is required at the method level we might be happy to consider only attacks which affects a method at a time. Otherwise the method is no longer extractable from the whole program, which can be considered as a form of protection against illicit use. However most attacks aims at hiding the copyright so that masked signature redistribution in the large should be considered harmful.

Counter-attacks may be classified as follows:

*Light-weight counter-attacks* essentially consist in improving the general-purpose abstract signature extractor for routine use (or using several ones with different analysis strategies according to the possible obfuscation strategies).

*Heavy-weight counter-attacks* may need human help and the subject program in order to built a specific abstract signature extractor for a case study (e.g. to prove copyright infringement).

## 9.1 Subtractive Attacks and Tamper-Proofing Counter-Attack

One can easily design a dependence analysis to discover which local variables of the method will have no effect on the method computation. The auxiliary variables `W` and `T` could be located in this way and the corresponding embedding instructions eliminated by program dependence analysis and slicing.

Classical tamper-proofing methods can detect if the program has been altered and cause the program to fail when. tampering is evident [4]. They can therefore be used to prevent subtractive attacks. A simple tamper-proofing method to avoid automatic subtractive attacks consists in creating dependencies between the subject program and the inserted code. We consider three examples:

1. values can be allocated on the heap instead of in variables which make live/dead variable analysis much more difficult;

2. one may, on one hand, have the random values $a$ and $b$ be chosen as values of the program integer constants, have $c$ be computed in terms of $a$ and $b$ and then $a$ recomputed back in terms of $b$ and $c$ and $b$ recomputed back in terms of $a$ and $c$. This false dependency could only be discovered by symbolic computation which is beyond the scope of most compilers and obfuscators;

3. since the static analysis does not take tests into account, opaque predicates [1, 6, 14] can be used to anchor the stegomark statements inlaid in the stegoprogram by creating interferences between the two which spuriousness or genuineness is computationally difficult, if not impossible to detect. Let `W` be an integer variable used in the stegomark, `V` be a variable used in the subject program, let $B(\mathtt{V}, \mathtt{W})$ be an opaque predicate which is always false (like $B(x, y) = 7y^2 - 1 = x^2$ where $x, y \in \mathtt{Z}$ [1]) and $g(\mathtt{V}, \mathtt{W})$ be any integer expression depending upon `V` and `W`. Then a stegomark statement of the form `W` = $f(\mathtt{W})$ can be anchored in the stegoprogram in the form:
   `if (`$B(\mathtt{V}, \mathtt{W})$`) { W =` $f(\mathtt{W})$`; V =` $g(\mathtt{V}, \mathtt{W})$ `}`

Since the value of the subject program variable `V` depends upon the value of the stegovariable `W`, it is hardly possible for an obfuscator to determine that the opaque predicate $B$ is always false and so the value of `W` is thought to be indispensable so that the stegomark cannot be eliminate. If this unexecuted code can be located by run-time observations, then variants can use an opaque predicate $B$ which is not identically false as in:

```
if (B) { V' = V; W = f(W); V = g(V, V', W) }
...
if (B) { V = V' }
```

Just in case, we propose a more advanced and original method to create a dependence between the stegomark and that of the subject method. For that purpose, we can use properties of the stegovariable `W` which hold in the standard semantics (more precisely signed 32 bits arithmetic). This is possible, for example, when the stegokey $n$ is a power of 2. Indeed, assume that $n = 2^k$. Then, we have:

$$\mathtt{W} = v + a.2^k \text{ in } \mathtt{Z} .$$

We also have, always in `Z`, that, for all $j \leq k$ :

$$\mathtt{W} \,\%\, 2^j = v \,\%\, 2^j$$

where $x\%y$ denotes the operation returning the rest of the euclidean division of $x$ by $y$. Observe that this property remains trivially

true in $\mathbb{Z}/_{2^{32}\mathbb{Z}}$ which is the domain of value of integer variables in Java™. We can therefore use these arithmetic properties to modify the computations of the subject method in which the stegomark is inlaid. For example, assume that $n = 2^{16}$ and that $v = 18$. Then whichever the value of the variable `W` is, we always have:

$$\texttt{W}\,\%\,4 = 2\ .$$

If, for example, the constant `1` appears explicitly in the subject program, then it can be replaced by $\texttt{W}\,\%\,4 - 1$. Now, if the variable `W` is dynamically modified using the techniques developed in Sec. 4, the variable `W` takes values during program execution which apparently stochastic, but have a hidden invariant which is used for stegomark anchoring. The stegoprogram thus modified preserves its original concrete semantics but would be irreversibly damaged if the stegomark is eliminated. An involved static [11] or dynamic analysis of the behaviour of the variable `W` is necessary to detect the invariant on which relies the dissimulation. Moreover, such constant dissimulations can be automatically generated at random points of the program.

## 9.2 Subtractive Attacks on Low Stealthiness and Counter-Attacks

***Static Attack on Low Stealthiness:*** As shown in the example of Sec. 5.2, this watermarking scheme often results in very unusual integer literal constants being inserted into the program. Literal constants with 5 or 6 digits as in the example can be extremely rare in real programs. An anonymous referee collected and classified all the integer literals from some 600 Java programs, 1.4 million lines in all and observed that 80% of all literal integers are between 0-99, 95% are between 0 and 999, 92% are powers of two or powers of two plus or minus 1. Because of this lack of stealthiness, the anonymous referee suggested that an attacker could expect to be able to locate the watermark code simply by looking for large literal constants. A light-weight counter-attack would be to compute large constants in term of small ones privileging powers of two or powers of two plus or minus 1 in this computation. A diversion would consists is spreading large integers elsewhere in the code, may be with dependences so that there elimination would make the code inoperative.

In the same line one can try to detect the iterations of polynomials of degree greater than or equal to 2, which are operations rarely occuring in practice.

When the stegokey $n = 2^k$ is a power of 2, a static attack consists in extracting modulo $n'$ where $n'$ is is the greatest power of 2 dividing $i = Q'(P(1))$. This yields $c' = P(1) \bmod n'$ such that the value of the signature $c$ corresponds to the $k$ lower bits of $c'$. Since $k$ is unknown, this provides only partial information on $c$, maybe too much for this choice to be considered safe. This shows that in all cases the stegosignature $c$ should be an encryption of the information not to be revealed.

In all cases, an obvious counter-attack on low stealthiness would be to hide the stegomark using a non-standard concrete semantics for signature extraction.

***Dynamic Attack on Low Stealthiness:*** In Sec. 4.2, we observed that the variable `W` will take on values that are "stochastic". An anonymous referee suggested to monitor the program execution to look for integer variables updated within loops whose values are random. A light-weight counter-attack consists in making the stegomark seldom executed together with pseudo-random number generators spread in the program for diversion. Other appropriate datatype obfuscations are considered in Sec. 9.5, including using a nonstandard semantics of (e.g. heap-allocated) floating point numbers for which such dynamic attacks would be much harder.

***Dynamic Attack on Unusual Variable Values:*** An anonymous referee suggested to monitor the execution to keep track of the sequence of values $i_0, i_1, \ldots, i_k$ successively taken by all integer variables `I` of the program. If `I` is a watermark variable then $n_j | g$ for some $j \in [1, \ell]$ where $g = \gcd(i_1 - i_0, i_2 - i_0, \ldots, i_k - i_0)$. A large $g$ is an indication that `I` might be a watermark variable and provides information on $n_j$ for factoring.

A light-weight counter-attack consists in exploiting the randomness in tests so that some assignments to watermark variables are rarely executed whence leading to a small $k$. Extraction is unchanged since tests are ignored. Another counter-attack is to rely on a non-standard semantics for the static analysis as considered e.g. in Sec. 5.3 since the above reasoning assumes the concrete values of watermark variables that are abstracted for extraction to be their execution values.

## 9.3 Reinterpretation and Counter-Attack

Reinterpretation or table interpretation consists in reencoding the program for a different virtual machine code. If the virtual machine specification is secret then again signatureless redistribution is impossible in the large whence might be considered harmless.

Otherwise this may require the redesign of the abstract interpreter used for signature extraction in order to take the virtual machine code into account. This might require rewriting the abstract signature extractor for all (known) Java™ virtual machines, an obviously heavy weight counter-attack.

## 9.4 Control Obfuscation and Counter-Attack

The objective of control obfuscation is to obscure the control flow without changing what the code does at runtime. Typically, selection and looping constructs are changed so that they no longer have a direct Java™ source code equivalent. Let us consider several classical such transformations.

***Sequential Code Reordering and Counter-Attack:*** Reordering of the code sequential composition, test and loop statements must preserve the order in which the elementary statements are executed in a method (unless this order is irrelevant) and so the static analysis of Sec. 5.1, which does not take the control structure of the method into account, is insensible to this transformation. If necessary, `goto` statements can easily be handled in static analysis.

***Proceduralization and Counter-Attack:*** Conversion of static to procedural data (make a procedure to compute a value instead of original constant).

Constant propagation can be trivially extended from the intraprocedural case considered here for stegosignature extraction to the interprocedural case [18], in which case the extraction technique remains the same.

***Outlining and Counter-Attack:*** Splitting a method into several disjoint methods (e.g. by inlining and different reproceduralization).

Again the counter-attack is interprocedural constant propagation.

***Parallelization and Counter-Attack:*** Conversion of sequential to parallel programs.

Again the transformation must preserve the order in which the elementary statements are executed so the static analysis of Sec. 5.1 is insensible to this transformation. Otherwise, the static analysis can be extended to parallel programs (see e.g. [8]).

## 9.5 Data Obfuscation and Counter-Attack

Another form of obfuscation is data obfuscation where the program global, local and heap data are reallocated in more complex structures. We now consider a few examples.

***Globalization and Counter-Attack:*** Replaces all or some local variables into global variables.

This might be easily taken into account by the analyser which could also consider global integer variables, either all possible global variables or only the necessary ones, which may not be very difficult to detect for those possessing the subject code (the list of such global variables might be a parameter of the static analysis).

***Built-in Datatype Reallocation and Counter-Attack:*** Put all data in arrays or heap allocated structures. First note that the absence of simple datatype variables is quite suspect. But not moving all simple variables means that the ones used to hide watermarks might be omitted. Using arrays means that constant propagation can detect indices designating simple variables so heap allocated structures should be preferred. But then note that if the obfuscator is likely to create a small dynamically allocated structure so that its shape might be easily determined by analysis algorithms using a threshold widening [17]. Otherwise cache behaviour and additional garbage collection might severely worsen the program performance.

Then note that this idea might be used to hide the stegovariables in dynamically allocated data structures which could hardly be modified by obfuscating by fear of modifying the program semantics but for which the signature extraction would be possible by designing analysers specifically for the type of structure which is used (e.g. balanced trees [9]).

***Built-in Datatype Obfuscation and Counter-Attack:*** Obfuscating built-in datatypes (such as integers and strings) by variable splitting and merging. To do so, obfuscators routinely use the algebraic law of integer arithmetic to transform the code. For a trivial example, consider the assignment:

```
e=35538;
```

which can be transformed into:

```
f=71077; e=(f-1)/2;
```

Since the integer division is not used in the stegomark, the static analyser need not implement the abstract version of integer division, which will be simply ignored, whence obliterating the signature extraction.

An obvious riposte is for the static analyser to anticipate all such possible obfuscating transformations and to:

1. use non-standard semantics which are invariant under such transformations (which, e.g. might not be the case when interpreting modulo arithmetic as integer arithmetic);

2. implement all abstract operators corresponding to the concrete operators that can be used for program transformation by obfuscators;

3. obfuscate the code in order to prevent further obfuscations.

A complementary solution consists in relying on program constructs for which data and operation transformation is very difficult. This is the case for example for floating point arithmetic which does not satisfy the usual mathematical identities (such as associativity, commutativity, etc) which are valid for the reals. It follows that obfuscators will have more difficulties to modify all floating point operations, except trivially.

Consequently, a simple riposte to obfuscation is to implement the stegomark with floating point arithmetic. Care must be taken to prove the absence of overflow (since floating point arithmetic is not modular) or to catch all potential exceptions that can be raised in the stegomark and to annihilate their effect. It follows that the stegomark does not perturb the normal stegoprogram execution.

Now in the non-standard semantics which is used for signature extraction, and therefore in the abstract semantics for the static analysis, all floating point operations of the stegoprogram can be interpreted as integer operations. The translation is simply one to one for arithmetic operators. Floating point constants must be converted into integers (may be up to some secret factor).

# 10. Implementation

The abstract software watermarker that we have designed and used for our experimentation is based on SOOT [20] which is a static analyser generator for Java™ itself written in Java™.

The SOOT optimizing framework offers different possible intermediate representations of Java™ source programs. For simplicity, we assume that the subject program $P \in \mathcal{P}$ is represented in Jimple style [20] , that is unstructured stackless 3-address code using typed auxiliary variables. Moreover, we can also assume that the `jsr` bytecode has been eliminated and the intermediate code is organized as a control flow graph of basic blocks. A decompiler is necessary to see the Jimple code in Java™ form.

The stegosignature embedder is implemented as specified in Sec. 4. The implementation of the extractor essentially amounts to that of the basic abstract domain, as well as the abstract environments, of the corresponding basic abstract operations and those used in the abstract transformer in SOOT. Then SOOT can generate automatically the static analyser described in Sec. 5.1. Finally, the abstract software watermarker has essentially to maintain a database of owners of stegokeys and corresponding stegosignatures and provides an elementary user interface.

# 11. Experiments

***Efficiency:*** The abstract software watermarking takes no additional developer time (but to choose which methods should be watermarked, to choose a stegosignature and to submit the subject software to the automatic stegosignature embedder). The recognition time to extract the stegosignature is comparable to compilation time and so is efficient.

The runtime costs are also negligible since for medium and large programs we could not observe significant modifications in the required memory and computation time resources.

***Robustness:*** We have conducted several experiments which consist in watermarking one method in a class, then in obfuscating the class with Java™ obfuscators (`Jcloak`™ and `Zelix klassmaster`) and then in extracting the signature from the obfuscated class. After a few improvements of the static analyser as described in Sec. 9, these obfuscators mainly using name obfuscation, flow obfuscation and string encryption could not disarm the

stegosignature extraction. Consequently, obfuscators can be used after signature embedding to obscure the work of stegoanalysts.

## 12. Conclusion

We have proposed a new class of software watermarking and fingerprinting methods called *abstract software watermarking*. The key idea is to anchor a stegomark in the program, that is statements which static analysis will reveal the stegosignatures. We exemplified an instance based on modular constant propagation parametrised by a secret stegokey, which is equivalent to infinitely many distinct instances of the abstract watermarker. The key idea is that the stegosignature extraction is neither static (it is based on the semantics of the program not on its syntax), nor dynamic (program execution does not reveal the stegosignature) but abstract (the stegosignature is revealed by abstract interpretation of a (may be non-standard) state or trace-based collecting semantics of the program). Since static analysis is undecidable (even for simple analyses like constant detection), the static analyser which is used for extraction can be designed to be involved enough so that extraction is impossible if the extraction algorithm is not perfectly known. Even if the signature extractor is made public, it is still possible to use abstract domains parametrised by secret stegokeys which make signature extraction computationally hard, if not impossible.

It is clear that stegoanalysis against this new class of abstract software watermarkers will improve. In response, the abstract software watermarking framework allows considering more sophisticated abstract interpretation-based static analysers thus making stegoattacks even more difficult. As is the case in cryptography, the rivalry between watermarkers and attackers may be endless and the source of much progress.

**Acknowledgements:** This work was supported by the RNRT ("Réseau National de Recherche en Télécommunications" of the french "Ministère de la Recherche" and the "Ministère de l'Économie, des Finances et de l'Industrie"), project n° 95 "Tatouage électronique sémantique de Code Mobile Java", 1999–2002. We thank J. D. GUTTMAN for his help with Sec. 6.1 and for inspiring Sec. 6.2, M. RIGUIDEL for bringing our attention to software watermarking, A. VENET for his participation in the early stage of the project, the Sable Research Group at McGill University, Montreal, Canada, in particular L. HENDREN, P. LAM and F. QIAN, for their help in the use of SOOT, especially during P. COUSOT's visit at McGill in September 2000, B. BLANCHET, J. FERET, A. MINÉ, D. MONNIAUX, X. RIVAL and the anonymous referees for their shrewd comments.

## 13. References

[1] ARBOIT, G. A method for watermarking Java™ programs via opaque predicates. In *Proc. Int. Conf. Electronic Commerce Research (ICECR-5)* (Montreal, CA, 23–27 Oct. 2002).

[2] BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S., AND YANG, K. On the (im)possibility of obfuscating programs. In *Proc. CRYPTO '2001, Santa Barbara, CA, LNCS 2139* (19–23 Aug. 2001), J. Kilian, Ed., Springer, 1–18.

[3] COLLBERG, C., AND THOMBORSON, C. Software watermarking: Models and dynamic embeddings. In *24th POPL* (San Antonio, TX, 20–22 Jan. 1997), ACM Press, 311–324.

[4] COLLBERG, C., AND THOMBORSON, C. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *IEEE Trans. Software Engrg. 28*, 8 (Aug. 2002), 735–746.

[5] COLLBERG, C., THOMBORSON, C., AND LOW, D. Breaking abstractions and unstructuring data structures. In *Proc. 1998 ICCL* (Chicago, IL, 14–16 May 1998), IEEE Comp. Soc. Press, 28–38.

[6] COLLBERG, C., THOMBORSON, C., AND LOW, D. Manufacturing cheap, resilient, and stealthy opaque constructs. In *25th POPL* (San Diego, CA, Jan. 1998), 184–196.

[7] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *6th POPL* (San Antonio, TX, 1979), ACM Press, 269–282.

[8] COUSOT, P., AND COUSOT, R. Invariance proof methods and analysis techniques for parallel programs. In *Automatic Program Construction Techniques*, A. Biermann, G. Guiho, and Y. Kodratoff, Eds. Macmillan, 1984, ch. 12, 243–271.

[9] GHIYA, R., AND HENDREN, L. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *23rd POPL* (St. Petersburg Beach, FL, 1996), ACM Press, 1–15.

[10] GOSLER, J. Software protection: Myth or reality? In *Proc. Advances in Cryptology — CRYPTO'85, LNCS 218* (Santa Barbara, CA, 18-22 Aug. 1985, 1986), H. Williams, Ed., Springer, 140–157.

[11] GRANGER, P. Static analysis of arithmetical congruences. *Int. J. Comput. Math. 30* (1989), 165–190.

[12] GUTTMAN, J. D. Private communication. 21 Jan. 2003.

[13] KILDALL, G. A unified approach to global program optimization. In *1st POPL* (Boston, MA, Oct. 1973), ACMpress, 194–206.

[14] MONDEN, A., IIDA, H., MATSUMOTO, K., INOUE, K., AND TORII, K. A practical method for watermarking Java™ programs. In *24th IEEE Computer Software and Applications Conf. , Compsac'2000* (Taipei, Taiwan, 25–29 Oct. 2000), 191–197.

[15] NAGRA, J., COLLBERG, C., AND THOMBORSON, C. A functional taxonomy for software watermarking. In *25th Australasian Computer Science Conf. (ACSC '2002)* (Melbourne, Australia, Jan. 2002), M. J. Oudshoorn, Ed., Conferences in Research and Practice in Information Technology, ACS.

[16] PALSBERG, J., KRISHNASWAMY, S., KWON, M., MA, D., SHAO, Q., AND ZHANG, Y. Experience with software watermarking. In *Proc. 16th ACSAC '00, New Orleans, LA* (11–15 Dec. 2000), IEEE Comp. Soc. Press.

[17] SAGIV, M., REPS, T., AND WILHELM, R. Shape analysis. In *Proc. Int. Conf. CC '2000, LNCS 1781* (Berlin, DE, 25 Mar. – 2 Apr. 2000), D. A. Watt, Ed., Springer, 1–17.

[18] SAGIV, M., REPS, T., AND HORWITZ, S. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoret. Comput. Sci. 167*, 1&2 (1996), 131–170.

[19] STERN, J., HACHEZ, G., KOEUNE, F., AND QUISQUATER, J.-J. Robust object watermarking: Application to code. In *Proc. 3rd Int. Work. on Information Hiding, IH '99* (Dresden, DE, 29 Sep. – 1 Oct. 1999), A.Pfitzmann, Ed., vol. 1768 of *LNCS*, Springer, 368–378.

[20] VALLÉE-RAI, R., HENDREN, L., SUNDARESAN, V., LAM, P., GAGNON, É., AND CO, P. Soot — a Java™ optimization framework. In *CASCON '99 (IBM Center for Advanced Studies Conference)* (Toronto, Ontario, CA, 8–11 Nov. 1999), 125–135.