Opaque Predicates Detection by Abstract Interpretation

Mila Dalla Preda[1], Matias Madou[2], Koen De Bosschere[2], and Roberto Giacobazzi[1]

[1] Department of Computer Science
University of Verona, Italy
dallapre@sci.univr.it,
roberto.giacobazzi@.univr.it
[2] Electronics and Information Systems Department
Ghent University, Belgium
{mmadou, kdb}@elis.UGent.be

Abstract. Code obfuscation and software watermarking are well known techniques designed to prevent the illegal reuse of software. Code obfuscation prevents malicious reverse engineering, while software watermarking protects code from piracy. An interesting class of algorithms for code obfuscation and software watermarking relies on the insertion of opaque predicates. It turns out that attackers based on a dynamic or an hybrid static-dynamic approach are either not precise or time consuming in eliminating opaque predicates. We present an abstract interpretation-based methodology for removing opaque predicates from programs. Abstract interpretation provides the right framework for proving the correctness of our approach, together with a general methodology for designing efficient attackers for a relevant class of opaque predicates. Experimental evaluations show that abstract interpretation based attacks significantly reduce the time needed to eliminate opaque predicates.

1 Introduction

The aim of *malicious reverse engineering* of software is to understand the inner workings of programs in order to identify vulnerabilities, to make unauthorized modifications or to steal the intellectual property of software. *Code obfuscation* is a well-known low cost approach to prevent malicious reverse engineering of software [2, 3]. The basic idea of code obfuscation is to transform programs so that the obfuscated programs are so difficult to understand that reverse engineering becomes too expensive in terms of resources or time. *Software piracy* refers to the illegal reproduction and distribution of software applications, whether for business or personal use. The aim of *software watermarking* is to dissuade illegal copying and reseal of programs. Software watermarking is a program transformation technique that embeds a signature into the software in order to encode some identifying information about it [4, 22].

1.1 The Problem

A predicate is opaque if its value is known a priori to a program transformation, while it is difficult for attackers to deduce it [2]. Opaque predicates can be used both for obfuscating and watermarking programs. In the case of code obfuscation, a class of obfuscating transformations known as control code obfuscators act by masking software control flow. Control code obfuscators often rely on inserting opaque predicates. Consider for example the insertion of a branch instruction controlled by an opaque predicate that always evaluates $true$, i.e., the $true$ path is always followed. Attackers are not aware of the constantly $true$ value of the opaque predicate, and have to take into account both $true$ and $false$ paths. On the other side, Monden et al. [22] store the watermark in a piece of dead code and then they make the watermark potentially reachable by inserting a true opaque predicate whose false branch transfers the control to the dead code containing the watermark. Therefore, a static analysis-based dead code removal does not eliminate the watermark, while the dead code itself is never executed. A different approach by Myles and Collberg [23] instead encodes the watermark in the constants used in opaque predicates. The resilience of an opaque predicate to attacks measures the resilience of the corresponding obfuscating/watermarking transformation. Here, we consider opaque predicates from number theory [1, 5, 23] such as $\forall x \in \mathbb{Z} : n|f(x)$, i.e., the function f always returns a multiple of n. More in general, we consider opaque predicates $\forall x \in \mathbb{Z} : f(x) \subseteq P$, i.e., the result of the function f always satisfies the property P. An attacker is a malicious user that wants to reverse engineer or copy a program for unlawful purposes, thus to succeed it has to defeat expected software protection techniques such as opaque predicate insertion. Once an opaque predicate is inserted in a program, it is possible to further protect the code using transformations meant to mask the opaque predicate itself. For example, hiding constant values by use of address computations or using bit-level operations to hide arithmetic manipulations are obfuscating transformations that mask the inserted opaque predicates. The de-obfuscation of these additional transformations and the opaque predicates detection are problems that can be studied independently. In the following we study a general and efficient methodology for disclosing opaque predicates, assuming that potential additional transformations have already been handled. We introduce a novel and efficient methodology of attack, based on Cousot and Cousot's abstract interpretation technique [7, 9], for eliminating opaque predicates. The present approach builds over the semantics-based view to code obfuscation introduced in [10, 11].

1.2 Main Results

We analyze two different approaches to opaque predicates detection. The first one is based on purely dynamic information, while the second one is based on hybrid static/dynamic information [16]. Experimental evaluations on a limited set of inputs show that a dynamic attack removes any opaque predicate, but it has the drawback of classifying many predicates as opaque, while they are not. Thus, dynamic attacks do not provide a trustful solution. Randomized algorithm may be used to eliminate opaque predicates, in this case the probability of precisely detecting an opaque predicate can be increased by augmenting the number of tries [14]. However randomized algorithms do not give an always trustful solution, but an answer that has an high probability of being precise. On the other hand, experimental evaluations on hybrid static/dynamic attacks show that breaking a single opaque predicate is rather time consuming, and may become unfeasible. We then introduce a novel methodology, based on formal program semantics and semantics approximation by abstract interpretation, to detect and then eliminate opaque predicates. Experimental evaluations show the efficiency of this new method of attack.

Attackers are malicious users that observe the behavior of the obfuscated program at different levels of abstraction with respect to the real program execution. The basic idea is to model attackers as abstract interpretations of the concrete program behaviour, i.e., the concrete program semantics. In this framework, an attacker is able to break an opaque predicate when the abstract detection of the opaque predicate is equivalent to its concrete detection. For opaque predicates as $\forall x \in \mathbb{Z} : n|f(x)$ and $\forall x \in \mathbb{Z} : f(x) \subseteq P$, this can be formalized as a completeness property of the underlying abstraction with respect to the function f. Completeness for an abstraction A with respect to some semantic function f means that no loss of precision is accumulated in the abstract computation of f on A with respect to its concrete computation. Abstract interpretation provides a systematic methodology for minimally refining an abstraction in order to make it complete for a given function. Thus, it turns out that completeness domain refinements provide here a systematic de-obfuscation technique that drives the design of abstractions, i.e., attackers, for disclosing opaque predicates.

2 Background

Notation. If $f : X^n \rightarrow Y$ is any n-ary function then its pointwise extension $f^p : \wp(X)^n \rightarrow \wp(Y)$ to the powerset is defined as $f^p(S_1, ..., S_n) \stackrel{\text{def}}{=} \{f(x_1, ..., x_n) \mid 1 \leq i \leq n,\ x_i \in S_i\}$. $\langle L, \leq, \vee, \wedge, \top, \bot\rangle$ denotes a complete lattice with ordering $\leq$, least upper bound (*lub*) $\vee$, greatest lower bound (*glb*) $\wedge$, greatest element $\top$ and least element $\bot$. Given an ordered set L the downward closure of $S \subseteq L$ is $\downarrow S \stackrel{\text{def}}{=} \{x \in L | \exists y \in S. x \leq y\}$, while the upward closure $\uparrow$ is dually defined. For $x \in L$, $\downarrow x$ is a shorthand for $\downarrow \{x\}$. Given $S \subseteq L$, $max(S) \stackrel{\text{def}}{=} \{x \in S \mid \forall y \in S. x \leq y \Rightarrow x = y\}$ is the set of maximal elements of S. Given any two functions $f, g : X \rightarrow L$, $f \sqsubseteq g$ denotes pointwise ordering, namely for any $x \in X$, $f(x) \leq g(x)$.

Abstract Interpretation. The basic idea of abstract interpretation is that the program behaviour at different levels of abstraction is an approximation of its formal semantics. The (concrete) semantics of a program is computed on the (concrete) domain $\langle C, \leq_C\rangle$, i.e., a complete lattice which models the values computed by programs. The partial ordering $\leq_C$ models relative precision between concrete values. An abstract domain $\langle A, \leq_A\rangle$ is a complete lattice which encodes an approximation of concrete program values. Abstract domains can be related to each other w.r.t. their relative degree of precision. Abstract domains are specified either by Galois connections (GCs), i.e., adjunctions, or by (upper) closures operators [7, 9]. Two complete lattices C and A form a Galois connection (C, α, γ, A), when $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ form an adjunction, namely $\forall a \in A, \forall c \in C : \ \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. α and γ are called, respectively, abstraction and concretization maps. An (upper) closure operator on C, or simply a closure, is an operator $\rho : C \rightarrow C$ which is monotone, idempotent, and extensive. We denote by $uco(C)$ the set of closures on C. When C is a complete lattice then $\langle uco(C), \sqsubseteq, \sqcap, \sqcup, \lambda x.\top, \lambda x.x\rangle$ is a complete lattice as well, where $\rho_1 \sqsubseteq \rho_2$ if and only if $\rho_2(C) \subseteq \rho_1(C)$, meaning that the abstract domain specified by ρ_1 is more precise than the abstract domain specified by ρ_2. Let us recall that each closure ρ is uniquely determined by the set of its fixpoints, given by its image $\rho(C)$. A set $X \subseteq C$ is the set of fixpoints of a closure operator if and only if X is a Moore family of C, i.e., $X = \mathcal{M}(X) \stackrel{\text{def}}{=} \{\wedge S | S \subseteq X\}$, where $\wedge\varnothing = \top \in \mathcal{M}(X)$. Given a GC (C, α, γ, A), $\rho = \gamma \circ \alpha$ is the closure corresponding to the abstract domain A.

Let (C, α, γ, A) be a GC, $f : C \to C$ a concrete function and $f^\sharp : A \to A$ an abstract function. $f^\sharp$ is a sound, i.e., correct, approximation of f if $\alpha \circ f \leq_A f^\sharp \circ \alpha$. When the soundness 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 in A. This means that no loss of precision is accumulated in the abstract computation through $f^\sharp$. Given $A \in uco(C)$ and a semantic function $f : C \to C$, the notation $f^A \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma$ denotes the best correct approximation of f in A [9]. It has been proved [12] that, given an abstraction A, there exists a complete approximation of $f : C \to C$ in A if and only if the best correct approximation f^A is complete. This means that completeness is an abstract domain property, namely that it depends on the structure of the abstract domain only. In particular, when an abstract domain is specified by a closure $\rho \in uco(C)$, we have that ρ is complete for f iff $\rho \circ f \circ \rho = \rho \circ f$ (soundness is instead encoded by $\rho \circ f \sqsubseteq \rho \circ f \circ \rho$). It turns out that an abstract domain $\rho \in uco(C)$ is complete for f if $\forall x \in \rho(C)$: $max(f^{-1}(\downarrow x)) \subseteq \rho(C)$, i.e., if ρ is closed under maximal inverse image of f. This leads to a systematic way for minimally refining an abstract domain in order to make it complete for a given semantic function [12]. The complete refinement of a domain ρ with respect to a function f is given by $\mathcal{R}_f(\rho) \stackrel{\text{def}}{=} gfp(\lambda X.\ \rho \sqcap \mathcal{M}(\cup_{y \in X} max(f^{-1}(\downarrow y))))$. It turns out that $\mathcal{R}_f(\rho)$ returns exactly the most abstract domain extending ρ and which is complete for f [12]. Thus, the completeness refinement adds the minimal amount of information needed to make the abstract domain complete. When f has more then one argument, for example when $f : C \times C \to C$, the maximal inverse image, i.e., $f^{-1}(x, y)$ is obtained by the union of the maximal inverse images of f for each fixed value of x and y [12]. For a set F of semantic functions, $\mathcal{R}_F(\rho)$ denotes the complete refinement of ρ for any function $f \in F$.

Opaque Predicates. A predicate is opaque if its outcome is known at embedding time, but it is hard for an attacker to deduce it [2, 3]. The basic idea is that the insertion of opaque predicates in a program makes the program control flow difficult for an attacker to analyze. Opaque predicates find interesting applications not only in code obfuscation techniques [15], but also in software watermarking [23] and tamper-proofing [24]. There exist two major kinds of opaque predicates: true opaque predicates, denoted by P^T, that always evaluate $true$, and false opaque predicates, denoted by P^F, that always evaluate $false$. Opaque predicates can be derived from number theory [3], alias analysis [2], concurrency [6], etc. We focus here on opaque predicates based on number theory of the form $\forall x \in \mathbb{Z} : n | f(x)$. These predicates are applied in some major software protection techniques as code obfuscation [3], software watermarking [23], tamper-proofing [24] and secure mobile agents [19]. Moreover, this class of opaque predicates is used in recent implementations such as PLTO [25] — a binary rewriting system that transforms a binary program preserving the functionality — LOCO [17] — a tool for binary obfuscating and de-obfuscating transformations — and

SANDMARK [5] — a tool for software watermarking, tamper proofing and code obfuscation of Java programs.

3 Dynamic Attack

Dynamic attackers execute programs with several (but of course not all) different inputs and observe the paths followed after each conditional jump. Thus, a dynamic attacker classifies a conditional jump as controlled by a false/true opaque predicate if, during these executions, the false/true path is always taken. Therefore, a dynamic attacker detects all the executed opaque predicates, but, due to the limited set of inputs considered, it may classify a predicate as opaque while it is not, called a *false negative*. Let us measure the false negative rate of a dynamic attacker. We execute the SPECint2000 benchmarks (without adding opaque predicates) with the reference inputs, and then we observe the conditional jumps. We use DIOTA[1] [18] to identify conditional jumps that always follow the true path, the false path or take both of them.

Table 1. Execution after conditional jumps

	% only flth	*% only jump*	*% both ways*
bzip2	42	23	35
crafty	24	19	57
gap	39	21	40
gcc	36	18	46
gzip	36	24	40
mcf	43	32	25
parser	29	14	57
perlbmk	45	23	33
twolf	39	21	40
vortex	59	26	15
vpr	42	21	38
average	***39***	***22***	***39***

The benchmarks are listed in Table 1. For each benchmark, the percentage of regular conditional jumps that look like false/true opaque predicates are annotated in the first/second column, while the percentage of regular conditional jumps are reported in the third column. Benchmarks do not contain opaque predicates, so that the opaque predicates detected by dynamic attack are false negatives. This experimental evaluations show that a dynamic attacker has an average of false negative rate of 39% and 22%, respectively for false and true opaque predicates. An attacker can improve these results using its knowledge of the program functionality in order to generate different inputs that are likely to execute different program paths. This will be very time consuming. Another way is to generate dynamic test data to improve the condition/decision coverage (CDC)[2]. For complex programs, the CDC is at most 58% [20], so 42% of all conditions will be seen as opaque predicates or dead code by the attacker which is of course incorrect. This leads us to conclude that dynamic attacks are too imprecise.

[1] DIOTA: a dynamic instrumentation tool which keeps a running program unaltered at its original location and generate instrumented code on the fly somewhere else.

[2] Condition/decision coverage measures the percentage of conditional jumps that are executed true at least once and false at least once.

4 Brute Force Attack

In this section we study an hybrid static/dynamic brute force attack acting on assembly basic blocks[3], where the instructions of the opaque predicate are statically identified (static phase) and are then executed on all possible inputs (dynamic phase). Let us consider the following opaque predicate $\forall x \in \mathbb{Z} : 2|(x^2 + x)$. Let us remark that the implementation of this opaque predicate decomposes the function $x^2 + x$ into elementary functions such as square x^2 and addition $x + y$. We make the assumption that the instructions (that is, elementary functions) corresponding to an opaque predicate are always grouped together, i.e., there are no program instructions between them. The static phase aims at identifying the instructions corresponding to an opaque predicate. Thus, for each conditional jump j the attack considers the instruction i immediately preceding j. The dynamic phase then checks whether i and j give rise to an opaque predicate. If this is the case the predicate is classified as opaque. Otherwise, the analysis proceeds upward by considering the next instruction preceding i, until an opaque predicate is found or the instructions in the basic block terminate. In this latter case, the predicate is not opaque. The computational effort, measured as number of steps, of the attack is $n^2 * (2^w)^r$, where n is the number of instructions of the opaque predicate, r is the number of registers and w is the width of the registers used by the opaque predicate. Consider for example the above true opaque predicate compiled for a 32-bit architecture. The predicate is executed with all possible 2^{32} inputs. This compiled code is then executed under the control of GDB, a well known open-source debugger[4], with all 2^{32} inputs. In particular $2|(x^2 + x)$ can be written in five x86 instructions, so that for this architecture the computational effort to break this opaque predicate will be $5^2 * 2^{64}$. During the hybrid attack, two variables are needed as input for the addition, so that there are at most 2 registers taken as input during the attack, i.e. r=2, and the width of these registers is 32 bits, i.e. $w = 32$.

It would be interesting to measure the time needed by this attack to detect an opaque predicate. Let us consider the opaque predicate $\forall x \in \mathbb{Z} : 2|(x + x)$ and measure the time needed to detect it. In assembly, this opaque predicate in a 16-bit environment consists of three instructions. The execution under control of GDB of these three assembly instructions with all 2^{16} inputs takes 8.83 seconds on a 1.6 GHz Pentium M processor with 1 GB of main memory running RedHat Fedora Core 3. In this experimental evaluation, the static phase has been performed by hand, meaning that the starting instruction of the opaque predicate was given. This leads us to conclude that the hybrid static/dynamic approach is precise although it is noticeably time consuming.

5 Breaking Opaqueness by Abstract Interpretation

We introduce an approach based on abstract interpretation for detecting opaque predicates. This novel technique leads to a formal characterization of a class of attackers that are able to break a specific type of commonly used opaque predicates, i.e.,

[3] A basic block is a sequence of instructions with a single entry point, single exit point, and no internal branches.

[4] http://www.gnu.org/software/gdb/

$\forall x \in \mathbb{Z} : n|f(x)$. This result can then be generalized to a wider class of opaque predicates, i.e., $\forall x \in \mathbb{Z} : f(x) \subseteq P$ where P is a generic property of integer numbers. In this case, we provide a methodology for designing efficient attackers. Experimental evaluations show how this abstract interpretation-based approach significantly reduces the computational effort of the attacker.

5.1 Modeling Attackers

Attackers have different precision degrees, according to the accuracy they have in observing program behaviours. We show that abstract interpretation turns out to be a suitable framework for modeling attackers and for classifying them according to their level of precision [10, 11]. Let $\langle \wp(\mathbb{Z}), \subseteq \rangle$ be the concrete domain for an integer program variable. An attacker can be modeled by an abstract domain $A \in uco(\wp(\mathbb{Z}))$, which may precisely represent the level of abstraction of an attacker. In the following, A denotes an abstract domain with partial ordering relation $\leq_A$, abstraction/concretization maps $\alpha_A : \wp(\mathbb{Z}) \to A$ and $\gamma_A : A \to \wp(\mathbb{Z})$. For example, the following well-known abstract domains *Sign* $= \{\mathbb{Z}, \mathbb{Z}_{\geq 0}, \mathbb{Z}_{\leq 0}, 0, \varnothing\}$ and *Parity* $= \{\mathbb{Z}, even, odd, \varnothing\}$ can model different attackers. Modeling attackers by abstract domains allows us to compare them with respect to their level of abstraction. Consider two attackers $A_1, A_2 \in uco(\wp(\mathbb{Z}))$. If A_2 is an abstraction of A_1, i.e., $A_1 \sqsubseteq A_2$, then the attacker A_1 is more precise (i.e., concrete) than the attacker A_2 in observing the obfuscated program. In our model, an attacker A breaks an opaque predicate when the abstract detection of the opaque predicate is equivalent to its concrete detection. Abstract domains can encode a significant approximation of the concrete domain. Accordingly, we will show that abstract detection of opaque predicates may result significantly simpler.

Attackers for Predicates $n|f(x)$. Let us consider numerical true opaque predicates of the form: $\forall x \in \mathbb{Z} : n|f(x)$, namely the function $f : \mathbb{Z} \to \mathbb{Z}$ always returns a value that is a multiple of $n \in \mathbb{Z}$. This class of opaque predicates is used in major obfuscating tools such as SANDMARK [5] and LOCO [17], and in the software watermarking algorithm by Arboit [1], recently implemented by Collberg and Myles [23].

In order to detect that the predicate $n|f(x)$ is opaque one needs to check the concrete test $\mathrm{CT}^f \overset{\text{def}}{=} \forall x \in \mathbb{Z} : f(x) \in n\mathbb{Z}$, where $n\mathbb{Z}$ denotes the set of integers that are multiples of $n \in \mathbb{Z}$. Our goal is to devise an abstract interpretation-based method which allows to perform the test of opaqueness for f on a suitable abstract domain. We are therefore interested in abstract domains which are able to represent precisely the property of being a multiple of n, i.e., abstract domains $A \in uco(\wp(\mathbb{Z}))$ such that there exists some $a_n \in A$ such that $\gamma_A(a_n) = n\mathbb{Z}$. Let $f^\sharp : A \to A$ be an abstract function that approximates f on A. Then, the *abstract test* on A is defined as follows:

$$AT_A^{f^\sharp} \overset{\text{def}}{=} \forall x \in \mathbb{Z} : f^\sharp(\alpha_A(\{x\})) \leq_A a_n$$

Definition 1. $AT_A^{f^\sharp}$ is *sound* (*complete*) when $AT_A^{f^\sharp} \Rightarrow CT^f$ ($AT_A^{f^\sharp} \Leftrightarrow CT^f$).

When $AT_A^{f^\sharp}$ is complete we also say that the attack $\langle A, f^\sharp \rangle$ (or simply A when $f^\sharp$ is clear from the context) *breaks* the opaque predicate $\forall x \in \mathbb{Z} : n|f(x)$.

Theorem 1. *Consider A such that there exists $a_n \in A$: $\gamma_A(a_n) = n\mathbb{Z}$, then:*

(1) *If $f^\sharp$ is sound approximation of f on the singletons, that is $\forall x \in \mathbb{Z}$, $\alpha_A(\{f(x)\}) \leq_A f^\sharp(\alpha_A(\{x\}))$, then $AT_A^{f^\sharp}$ is sound.*
(2) *If $f^\sharp$ is complete approximation of f on the singletons, that is $\forall x \in \mathbb{Z}$, $\alpha_A(\{f(x)\}) = f^\sharp(\alpha_A(\{x\}))$, then $AT_A^{f^\sharp}$ is complete.*

Thus, the key point is to design a suitable abstract domain A together with a complete approximation $f^\sharp$ of f.

Abstract Functions. We already observed in Section 4 that a function $f : \mathbb{Z} \to \mathbb{Z}$ is decomposed into elementary functions, i.e. assembly instructions within some basic block. Following the same approach, let us assume that the function f can be expressed as a composition of elementary functions, namely $f = \lambda x.h(g_1(x, ..., x), ..., g_k(x, ..., x))$ where $h : \mathbb{Z}^k \to \mathbb{Z}$ and $g_i : \mathbb{Z}^{n_i} \to \mathbb{Z}$. More in general, each g_i can be further decomposed into elementary functions. For example, $f(x) = x^2 + x$ is decomposed as $h(g_1(x), g_2(x))$ where $h(x, y) = x + y$, $g_1(x) = x^2$ and $g_2(x) = x$. Let us consider the pointwise extensions of the elementary functions, which are still denoted, with a slight abuse of notation, by $h : \wp(\mathbb{Z})^k \to \wp(\mathbb{Z})$ and $g_i : \wp(\mathbb{Z})^{n_i} \to \wp(\mathbb{Z})$, and let us denote their composition by $F \overset{\text{def}}{=} \lambda X.h(g_1(X, ..., X), ..., g_k(X, ..., X)) : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$. For example, for the above decomposition $f(x) = x^2 + x = h(g_1(x), g_2(x))$, we have that $F : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ is as follows: $F(X) = \{y^2 + z \mid y, z \in X\}$. Observe that F does not coincide with the pointwise extension f^p of f, e.g., $F(\{1, 2\}) = \{2, 3, 5, 6\}$ while $f^p(\{1, 2\}) = \{2, 6\}$. Let us also notice that F on singletons coincides with f, namely for any $x \in \mathbb{Z}$, $F(\{x\}) = f(x)$. Thus, the concrete test CT^f can be equivalently formulated as $\forall x \in \mathbb{Z} : F(\{x\}) \subseteq n\mathbb{Z}$.

Let $A \in uco(\wp(\mathbb{Z}))$ be an abstract domain such that there exists some $a_n \in A$ with $\gamma_A(a_n) = n\mathbb{Z}$. The attacker A approximates the computation of the function $F : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ in a step by step fashion, meaning that A approximates every elementary function composing F. Thus, the abstract function $F^\sharp : A \to A$ is defined as the composition of the best correct approximations h^A and g_i^A on A of the elementary functions, namely:

$$F^\sharp(a) \overset{\text{def}}{=} \alpha_A(h(\gamma_A(\alpha_A(g_1(\gamma_A(a), ..., \gamma_A(a)))), ..., \gamma_A(\alpha_A(g_k(\gamma_A(a), ..., \gamma_A(a))))))$$

When the abstract test $AT_A^{F^\sharp}$ for $F^\sharp$ on A holds, the attacker modeled by the abstract domain A classifies the predicate $n|f(x)$ as opaque. It turns out that $F^\sharp$ is a correct approximation of F on A, namely $\alpha_A \circ F \sqsubseteq_A F^\sharp \circ \alpha_A$, and this guarantees the soundness of the abstract test $AT_A^{F^\sharp}$.

Corollary 1. *$AT_A^{F^\sharp}$ is sound.*

Consider for example the opaque predicate $\forall x \in \mathbb{Z} : 3|(x^3 - x)$, and the abstract domain A_{3+} in the figure below. A_{3+} precisely represents the property of being a multiple of 3, i.e. $3\mathbb{Z}$, and its negation, i.e. $\mathbb{Z} \smallsetminus 3\mathbb{Z}$.

$$\begin{tikzcd}
& \mathbb{Z} \arrow[dl, dash] \arrow[dr, dash] & \\
3\mathbb{Z} \arrow[dr, dash] & & \mathbb{Z} \smallsetminus 3\mathbb{Z} \arrow[dl, dash] \\
& \varnothing &
\end{tikzcd}$$

In this case, $f(x) = x^3 - x = h(g_1(x), g_2(x))$ where $h(x, y) = x - y$, $g_1(x) = x^3$ and $g_2(x) = x$, so that $F : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ is given by $F(X) = \{y^3 - z \mid y, z \in X\}$. Hence, it turns out that $F^\sharp(3\mathbb{Z}) = 3\mathbb{Z}$ while $F^\sharp(\mathbb{Z} \smallsetminus 3\mathbb{Z}) = \mathbb{Z}$. Here, the abstract test $AT^{F^\sharp}_{A_{3+}}$ is sound but not complete, because $F^\sharp : A_{3+} \to A_{3+}$ is a sound but not complete approximation of f on the singletons. In fact, for $\{2\} \in \wp(\mathbb{Z})$, it turns out that $\alpha_{A_{3+}}(\{f(2)\}) = \alpha_{A_{3+}}(\{6\}) = 3\mathbb{Z}$ while $F^\sharp(\alpha_{A_{3+}}(\{2\})) = F^\sharp(\mathbb{Z} \smallsetminus 3\mathbb{Z}) = \mathbb{Z}$. Thus the abstract test $AT^{F^\sharp}_{A_{3+}}$, i.e., $\forall x \in \mathbb{Z} : F^\sharp(\alpha_{A_{3+}}(\{x\})) \leq 3\mathbb{Z}$ does not hold even if CT^f does. Thus, in general $AT^{F^\sharp}_A$ is sound but not complete, meaning that the attacker $\langle A, F^\sharp \rangle$ is not able to break the opaque predicate $\forall x \in \mathbb{Z} : n|f(x)$.

Recall that abstract domain completeness is preserved by function composition [12], i.e. if an abstract domain is complete for f and g then A is complete for $f \circ g$ as well. As a consequence, if an abstract domain A is complete for the elementary functions h and g_i that decompose F then A is complete also for their composition F. It turns out that completeness of an abstract domain A w.r.t. the elementary functions composing F guarantees that the attacker A is able to break the opaque predicate $\forall x \in \mathbb{Z} : n|f(x)$.

Corollary 2. *If A is complete for the elementary functions h and g_i composing F then $\langle A, F^\sharp \rangle$ breaks the opaque predicate $\forall x \in \mathbb{Z} : n|f(x)$.*

Let us consider the opaque predicate $\forall x \in \mathbb{Z} : 3|(x^3 - x)$ and the abstract domain 3-$arity$ represented in the following figure.

$$\begin{tikzcd}
& \mathbb{Z} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
3\mathbb{Z} \arrow[dr, dash] & 1 + 3\mathbb{Z} \arrow[d, dash] & 2 + 3\mathbb{Z} \arrow[dl, dash] \\
& \varnothing &
\end{tikzcd}$$

The function $f(x) = x^3 - x$ is decomposed as $h(g_1(x), g_2(x))$ where $h(x, y) = x - y$, $g_1(x) = x^3$ and $g_2(x) = x$. It turns out that the abstract domain 3-$arity$ is complete for the pointwise extensions of h, g_1 and g_2, i.e. $\lambda\langle X, Y\rangle.X - Y$, $\lambda X.X^3$ and $\lambda X.X$, and therefore, by Corollary 2, the attacker 3-$arity$ is able to break the opaque predicate $\forall x \in \mathbb{Z} : 3|(x^3 + x)$.

Lemma 1. *3-$arity$ is complete for $\lambda X.X^3$, $\lambda X.X$ and $\lambda\langle X, Y\rangle.X - Y$.*

Experimental results. A prototype of the above described attack based on the abstract domain *Parity* has been implemented using LOCO [17], a x86 tool for obfuscation/deobfuscation transformations which is able to insert opaque predicates. This experimental evaluation has been conducted on the aforementioned 1.6 GHz Pentium M-based system. Each program of the SPECint2000 benchmark suite is obfuscated by inserting the following true opaque predicates: $\forall x \in \mathbb{Z} : 2|(x^2 + x)$ and $\forall x \in \mathbb{Z} : 2|(x + x)$.

It turns out that *Parity* is complete for addition, square and identity function, thus by Corollary 2, the abstract domain *Parity* models an attacker that is able to break these opaque predicates. In the obfuscating transformation each basic block of the input assembly program is split into two basic blocks. Then, LOCO checks whether the opaque predicate can be inserted between these two basic blocks: a liveness analysis is used here to ensure that no dependency is broken and that the obfuscated program is functionally equivalent to the original one. In particular, liveness analysis checks that the registers and the conditional flags affected by the opaque predicate are not live in the program point where the opaque predicate will be inserted. Moreover, our tool also checks by a standard constant propagation whether the registers associated to the opaque predicate are constant or not. If constant propagation detects that these are constant then the opaque predicate can be trivially broken and therefore is not inserted. Although liveness analysis and constant propagation are noticeably time-consuming, they are nevertheless necessary both to certificate functional equivalence between original and obfuscated program and to guarantee that the opaque predicate cannot be trivially broken by constant propagation. The algorithm used to detect opaque predicates is analogous to the brute force attack algorithm described in Section 4. Fig. 1 describes the basic block, by pseudo-code, which implements the opaque predicate $\forall x \in \mathbb{Z} : 2|(x^2 + x)$.

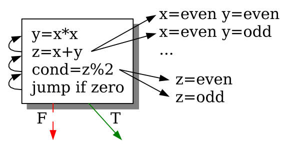

Fig. 1. Breaking $\forall x \in \mathbb{Z},\ 2|(x^2 + x)$

Let us describe how our de-obfuscation algorithm works. For each conditional jump j, `jump if zero` in the figure, we consider the instruction i which immediately precedes j, `cond=z%2` in the figure. The instructions j and i are abstractly executed on each value of the abstract domain (i.e. the attack). In the considered case of the attack modeled by *Parity*, both non-trivial values $even$ and odd are given as input to `cond=z%2`. When `z` evaluates to $even$, `cond` evaluates to 0 and therefore the true path is followed. On the other hand, when `z` is evaluated to *odd*, `cond` evaluates to 1 and the false path is taken. Thus, i does not give rise to an opaque predicate, so that we need to consider the instruction `z=x+y` which immediately precedes i. The instruction `z=x+y` is binary and therefore we need to consider all the values in *Parity* × *Parity*. This process is iterated until an opaque predicate is detected or the end of the basic block is reached. In our case, the opaque predicate is detected when the algorithm analyses the instruction `y=x*x` because whether `x` is evaluated to $even$ or odd the true path is taken. The number of computational steps needed for breaking one single opaque predicate by an attack based on an abstract domain A is $n^2 * d^r$, where n is the number of instructions composing the opaque predicate, r is the number of registers used by the opaque predicate and d is the number of abstract values in A. The reduction of the computational effort of the abstract interpretation-based attack with respect to the brute force attack can therefore be huge since the abstract domain can encode a very coarse approximation. In the considered example, the number of steps for detecting $\forall x \in \mathbb{Z} : 2|x + x$ through the abstract domain *Parity* results to be $3^2 * 2^2$. In fact, the opaque predicate consists of 3 instructions, uses 2 registers and *Parity* has 2 non-trivial abstract values. In Table 2 we show the results of the obfuscation/de-obfuscation process on the SPECint2000 benchmark suite. The first and second columns report respectively the number of opaque predicates inserted in each benchmark and the time needed for such obfuscation, while the third column lists the time needed to de-obfuscate. It turns out that the *Parity*-based de-obfuscation process is able to detect all the inserted opaque predicates. Let us recall that the brute force attack took 8.83 seconds to detect only one occurrence of the opaque predicate $\forall x \in \mathbb{Z} : 2|x + x$ in a 16-bit environment, while the abstract interpretation-based de-obfuscation attack took 8.13 seconds to de-obfuscate 66176 opaque predicates in a 32-bit environment.

Table 2. Timings of obfuscation and de-obfuscation

	# opaque pred	*time obf*	*time deobf*
bzip2	442	0.53	0.13
crafty	3298	35.18	0.47
gap	6659	8.59	1.02
gcc	28006	476.33	3.11
gzip	734	0.44	0.11
mcf	189	0.29	0.06
parser	2543	2.48	0.31
perlbmk	10255	42.71	1.23
twolf	3575	1.88	0.48
vortex	8269	8.79	0.91
vpr	2206	1.44	0.3
Total	**66176**	**578.66**	**8.13**

6 Designing Domains for Breaking Opaque Predicates

This section shows how the completeness domain refinements can be used to derive models of attackers which are able to break a given opaque predicate. Let us consider the opaque predicate $\forall x \in \mathbb{Z} : 3|(x^3 - x)$ and the attacker $A_3 \overset{\text{def}}{=} \{\mathbb{Z}, 3\mathbb{Z}\}$, that is the minimal abstract domain which represents precisely the property of being a multiple of 3. Recall that the function $f(x) = x^3 - x$ is decomposed as $h(g_1(x), g_2(x))$ where $h(x, y) = x - y$, $g_1(x) = x^3$ and $g_2(x) = x$. It turns out that A_3 is not able to break the above opaque predicate, since $F^\sharp : A_3 \to A_3$ is not a complete approximation of f on singletons. In fact, consider $\{2\} \in \wp(\mathbb{Z})$, it turns out that $\alpha_{A_3}(\{f(2)\}) = \alpha_{A_3}(\{6\}) = 3\mathbb{Z}$ while $F^\sharp(\alpha_{A_3}(\{2\})) = F^\sharp(\mathbb{Z}) = \mathbb{Z}$. Corollary 2 does apply here because A_3 is complete for g_1 and g_2 but not for h. However, as recalled in Section 2, completeness can be obtained by a domain refinement. We thus systematically transform A_3 by the completeness domain refinement w.r.t. $h = \lambda\langle X, Y\rangle.X - Y$. We obtain the abstract domain $\mathcal{R}_h(A_3)$ that models an attacker which is able to break $\forall x \in \mathbb{Z} : 3|(x^3 - x)$. As recalled in Section 2, the application of the completeness domain refinement adds to $A_{3\mathbb{Z}}$ the maximal inverse images under h of all its elements until a fixpoint is reached, that is for any fixed $X \subseteq \mathbb{Z}$ and a belonging to the current abstract domain, we iteratively add the following sets of integers: *max*$\{Z \subseteq \mathbb{Z} \mid Z - X \subseteq a\}$. It is not hard to verify that the following elements provide exactly the minimal amount information to add to A_3 in order to make it complete for h.

- if $X = \{0\}$ then: *max*$\{Z \subseteq \mathbb{Z} \mid Z - X \subseteq 3\mathbb{Z}\} = 3\mathbb{Z}$
- if $X = \{1\}$ then: *max*$\{Z \subseteq \mathbb{Z} \mid Z - X \subseteq 3\mathbb{Z}\} = 1 + 3\mathbb{Z}$
- if $X = \{2\}$ then: *max*$\{Z \subseteq \mathbb{Z} \mid Z - X \subseteq 3\mathbb{Z}\} = 2 + 3\mathbb{Z}$

Therefore, $\mathcal{R}_h(A_3) = \{\mathbb{Z}, 3\mathbb{Z}, 1+3\mathbb{Z}, 2+3\mathbb{Z}, \varnothing\} = 3\text{-}arity$. Let us notice that we were able to systematically obtain the attacker $3\text{-}arity$, which is able to break the opaque predicate, through a completeness refinement of the minimal abstract domain A_3.

It turns out that given $n \in \mathbb{N}$, the abstract domain $n\text{-}arity$, in the figure below, is complete for addition, difference and, for $k \in \mathbb{N}$, k-power (i.e., $\lambda X.X^k$). Therefore, by Corollary 2, the attacker $n\text{-}arity$ breaks the opaque predicates $\forall x \in \mathbb{Z}, n|f(x)$, where f is a polynomial function. Observe that the abstract domain $n\text{-}arity$ is an instance of Granger's domain of congruences [13].

Theorem 2. *The attacker $n\text{-}arity$ breaks all the opaque predicates of the following form: $\forall x \in \mathbb{Z}:\ n|f(x)$, where $f(x)$ is a polynomial function.*

$$\begin{tikzcd}
 & & \mathbb{Z} \arrow[dll, dash] \arrow[dl, dash] \arrow[d, dash] \arrow[drr, dash] & & \\
n\mathbb{Z} \arrow[drr, dash] & 1+n\mathbb{Z} \arrow[dr, dash] & 2+n\mathbb{Z} \arrow[d, dash] & \dots & (n-1)+n\mathbb{Z} \arrow[dll, dash] \\
 & & \bot & &
\end{tikzcd}$$

6.1 Breaking Opaque Predicates $P(f(x))$

Let us now consider the wider class $P(f(x))$ of opaque predicates where each predicate has the following form: $f(x) \subseteq P$, with $P \subseteq \mathbb{Z}$ and $f : \mathbb{Z} \to \mathbb{Z}$. It is possible to generalize the results of the previous sections, in particular Theorem 1, Corollary 1 and Corollary 2, to opaque predicates in $P(f(x))$. This is simply done by replacing the property $n\mathbb{Z}$ of being a multiple of n, with a general property P over integers. This allows us to provide a formal methodology for designing abstract domains that model attackers able to break opaque predicates in $P(f(x))$. Let $\forall x \in \mathbb{Z} : f(x) \subseteq P$ be an opaque predicate and let us consider the minimal abstract domain A_P that represents precisely the property P, i.e., $A_P \overset{\text{def}}{=} \{\mathbb{Z}, P\}$. As above, we assume that the function f can be expressed as a composition of elementary functions, namely $f = \lambda x.h(g_1(x, ..., x), ..., g_k(x, ..., x))$ where $h : \mathbb{Z}^k \to \mathbb{Z}$ and $g_i : \mathbb{Z}^{n_i} \to \mathbb{Z}$. Then, we compute the completeness domain refinement of A_P w.r.t. the set of elementary functions composing f, namely $\mathcal{R}_{h,g_1,...,g_k}(A_P)$. It turns out that the refined domain is able to break the opaque predicate $\forall x \in \mathbb{Z} : f(x) \subseteq P$.

Theorem 3. *The attacker modeled by the abstract domain $\mathcal{R}_{h,g_1,...,g_k}(A_P)$ breaks the opaque predicate $\forall x \in \mathbb{Z} : f(x) \subseteq P$.*

Thus, completeness domain refinements provide here a systematic methodology for designing attackers that are able to break opaque predicates of the form: $\forall x \in \mathbb{Z} : f(x) \subseteq P$.

The previous result is independent from the choice of the concrete domain $\mathbb{Z}$ and can be extended to a general domain of computation *Dom*.

Corollary 3. *Consider an opaque predicate $\forall x \in$ Dom: $f(x) \subseteq P$, where f : Dom $\rightarrow$ Dom, $f = h(g_1(x, ..., x), ..., g_k(x, ..., x))$, and $P \subseteq$ Dom. The abstract domain $\mathcal{R}_{h,g_1,...,g_k}(\{Dom, P\})$ is able to break opaque predicate.*

7 Conclusion and Future Work

In this work we propose an abstract interpretation-based approach to detect opaque predicates. It turns out that, considering opaque predicates of the form: $\forall x \in \mathbb{Z} : n|f(x)$, the ability of an attacker, i.e., an abstract domain, to break opaque predicates can be formalized as a completeness problem w.r.t. function f. Consequently completeness domain refinement can be used to derive efficient attackers. In particular it turns out that the abstract domain n-$arity$ breaks the opaque predicate $\forall x \in \mathbb{Z} : n|f(x)$, where n ranges over $\mathbb{N}$ and f is a polynomial function. This result is then generalized to a wider class of opaque predicates of the form $\forall x \in \mathbb{Z} : f(x) \subseteq P$, where the attacker able to break the opaque predicate is obtained by completeness refinement of the abstract domain $A_P = \{\mathbb{Z}, P\}$.

The insertion of an opaque predicate code creates a path that is never taken. Notice that when the false path of a true opaque predicate contains another opaque predicate the degree of obfuscation of the transformation increases. The two opaque predicates interact with each other, and this dependence adds more confusion in the understanding of the original control flow of the program. Thus the insertion of dependent opaque predicates can be seen as a novel obfuscation technique.

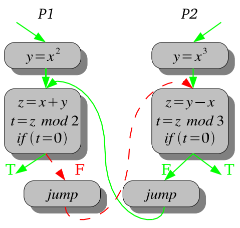

Fig. 2. Dependent opaque predicate

Consider for example the true opaque predicates $P1 : \forall x \in \mathbb{Z} : 2|(x^2 + x)$ and $P2 : \forall x \in \mathbb{Z} : 3|(x^3 - x)$ that interact with each other as depicted in the above figure. On the left-hand side we have the opaque predicate $P1$, while on the right-hand side we have $P2$, expressed in terms of elementary functions, i.e., assembly instructions. Observe that the false branch of predicate $P1$ enters the second basic block of predicate $P2$ and vice versa. The attacker modeled by the abstract domain *Parity* should be able to break opaque predicate $P1$. The problem is that *Parity* cannot break $P2$ and therefore we have an incoming edge on the second basic block of opaque predicate $P1$ coming from $P2$. This gives the idea of why we are no longer able to break opaque predicate $P1$ with the *Parity* domain. Therefore when there are opaque predicates that interact with each other the attacker needs to take into account these dependencies. Our guess is that a suitable attacker to handle this situation could probably be obtained by combining the abstract domains breaking the individual opaque predicates. The main problem is that one opaque predicate which is not breakable by our technique could protect breakable opaque predicates by interacting with these opaque predicates.

It would be also interesting to consider abstract domains that are more complex than the ones considered so far. Program properties that can be studied only on more complex domains could lead to the design of novel opaque predicates. Since these properties derive from a more complex analysis the corresponding opaque predicates should be more resilient to attacks. Consider for example the polyhedral abstract domain [8] and the abstract domain of octagons [21] for discovering properties of numerical variables.

Acknowledgments. The authors would like to thank the Fund for Scientific Research - Belgium - Flanders (FWO) and Ghent University for their financial support.

References

1. G. Arboit. A Method for Watermarking Java Programs via Opaque Predicates. In *Proc. Int. Conf. Electronic Commerce Research (ICECR-5)*, 2002.
2. C. Collberg, C. Thomborson and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proc. ACM POPL'98*, pp. 184-196, 1998.
3. C. Collberg, C. Thomborson and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, The University of Auckland, New Zealand, 1997.
4. C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn and M. Stepp. Dynamic Path-Based Software Watermarking. In *Proc. ACM PLDI'04*, pp. 107-118, 2004.
5. C. Collberg, G. Myles and A. Huntwork. SandMark - A Tool for Software Protection Research. *IEEE Security and Privacy*, 1(4):40-49, 2003.
6. C. Collberg. CSc620: Security through Obscurity. Handouts of a course available at: www.cs.arizona.edu/ collberg/Teaching/620/ 2002/Handouts/Handout-13.ps, 2002.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM POPL'77*, pp. 238–252, 1977.
8. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. ACM POPL'78*, pp. 84–97, 1978.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM POPL'79*, pp. 269–282, 1979.
10. M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *Proc. 32nd ICALP*, LNCS 3580, pp. 1325-1336, 2005.
11. M. Dalla Preda and R. Giacobazzi. Control Code Obfuscation by Abstract Interpretation. In *Proc. 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pp. 301-310, 2005.
12. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM.*, 47(2):361-416, 2000.
13. P. Granger. Static analysis of linear congruence equality among variables of a program. In *Proc. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'91)*, pp. 169-192, 1991.
14. J. Hormkovic. Algorithmics for Hard Problems. Springer-Verlag, 2002.
15. C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proc. 10th ACM Conference on Computer and Communications Security (CCS'03)*, 2003.

16. M. Madou, B. Anckaert, B. De Sutter and K. De Bosschere. Hybrid Static-Dynamic Attacks Against Software Protection Mechanisms. In *Proc. 5th ACM Workshop on Digital Rights Management (DRM'05)*, 2005.
17. M. Madou, L. Van Put and K. De Bosschere. Loco: An Interactive Code (De)Obfuscation tool. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*, 2006.
18. J. Maebe, M. Ronsse and K. De Bosschere. DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. In *Proc. 4th Workshop on Binary Translation (WBT'02)*, 2002.
19. A. Majumdar and C. Thomborson. Securing Mobile Agents Control Flow Using Opaque Predicates. In *Proc. 9th Int. Conf. Knowledge-Based Intelligent Information and Engineering Systems (KES'05)*, 2005.
20. C. Michael, G. McGraw, M. Schatz and C. Walton. Genetic Algorithms for Dynamic Test Data Generation. In *Proc. ASE'97*, pp. 307-308, 1997.
21. A. Minè. The octagon abstract domain. In *Proc. Analysis, Slicing and Transformation (AST'01)*, pp. 310-319, 2001.
22. A. Monden, H. Iida, K. Matsumoto, K. Inoue and K. Torii. A Practical Method for Watermarking Java Programs. In *Proc. 24th Computer Software and Applications Conference*, pp. 191-197, 2000.
23. G. Myles and C. Collberg. Software Watermarking via Opaque Predicates: Implementation, Analysis, and Attacks. In *Proc. Int. Conf. Electronic Commerce Research (ICECR-7)*, 2004.
24. J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao and Y. Zhang. Experience with Software Watermarking. In *Proc. 16th Annual Computer Security Applications Conference (ACSAC'00)*, pp. 308-316, 2000.
25. B. Schwarz, S. Debray and G. Andrews PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In *Proc. Workshop on Binary Translation (WBT'01)*, 2001.