Under consideration for publication in Formal Aspects of Computing

Code Obfuscation Against Abstraction Refinement Attacks

Roberto Bruni[1], Roberto Giacobazzi[2,3] and Roberta Gori[1]

[1] Dipartimento di Informatica, Università di Pisa
[2] Dipartimento di Informatica, Università di Verona
[3] IMDEA SW Institute, Spain

Abstract. Code protection technologies require anti reverse engineering transformations to obfuscate programs in such a way that tools and methods for program analysis become ineffective. We introduce the concept of model deformation inducing an effective code obfuscation against attacks performed by abstract model checking. This means complicating the model in such a way a high number of spurious traces are generated in any formal verification of the property to disclose about the system under attack. We transform the program model in order to make the removal of spurious counterexamples by abstraction refinement maximally inefficient. Because our approach is intended to defeat the fundamental abstraction refinement strategy, we are independent from the specific attack carried out by abstract model checking. A measure of the quality of the obfuscation obtained by model deformation is given together with a corresponding best obfuscation strategy for abstract model checking based on partition refinement.

Keywords: Code obfuscation, verification, model checking, refinement

1. Introduction

1.1. The scenario

Software systems are a strategic asset, which in addition to correctness deserves security and protection. This is particularly critical with the increase of mobile devices and ubiquitous computings, where the traditional black-box security model, with the attacker not able to see into the implementation system, is not adequate anymore. Code protection technologies are increasing their relevance due to the ubiquitous nature of modern untrusted environments where code runs. From home networks to consumer devices (e.g., mobile devices, web browsers, cloud, and IoT devices), the running environment cannot guarantee integrity and privacy. Existing techniques for software protection originated with the need to protect license-checking code in software, particularly in games or in IP protection. Sophisticated techniques, such as white-box (WB) cryptography and software watermarking, were developed to prevent adversaries from circumventing media anti-piracy protection in Digital Rights Management systems.
A WB attack model to a software system $\mathcal{S}$ assumes that the attacker has full access to all of the components of $\mathcal{S}$, i.e., $\mathcal{S}$ can be inspected, analysed, verified, reverse-engineered, or modified. The goal of the attack is to disclose properties of the run-time behaviour of $\mathcal{S}$. These can be a hidden watermark [31, 24, 11], a cryptographic key or an invariance property for disclosing program semantics and make correct reverse

Correspondence and offprint requests to: Roberta Gori email: roberta.gori@unipi.it

engineering [9]. Note that standard encryption is only partially applicable for protecting $\mathcal{S}$ in this scenario: The transformed code has to be executable while being protected. Protection is therefore implemented as *obfuscation* [8]. Essentially, an obfuscation is a compiler that transforms an input program p into a semantically equivalent one $\mathcal{O}(p)$ but harder to analyse and reverse engineer.
In many cases it is enough to guarantee that the attacker cannot disclose the information within a bounded amount of time and with limited resources available. This is the case if new releases of the program are issued frequently or if the information to be disclosed is some secret key whose validity is limited in time, e.g., when used in pay-per-view mobile entertainment and in streaming of live events. Here the goal of the code obfuscation is to prevent the attacker from disclosing some keys before the end of the event.
Impossibility results of virtual black-box (VBB) obfuscation [2], proves that there exists a fundamental difference between having black-box access to a function and having a WB access to a program that computes it, no matter how obfuscated: The program provides an intensional representation of the function, always disclosing information about its implementation, allowing reverse-engineering of code. Similarly to Rice's impossibility theorem [26] which did not dishearten the development of methods and tools for automatically proving program correctness, the impossibility of VBB obfuscation represents a major challenge in developing practical methods for hiding sensitive information in programs that guarantee secure non-disclosure, e.g., for a limited amount of time or against specific attacks.
The current state of the art in code protection by obfuscation is characterised by a scattered set of methods and commercial/open-source techniques employing often ad hoc transformations that complicate code yet keeping its functionality, see [9] for an excellent survey. Examples of obfuscating transformations include code flattening to remove control-flow graph structures, randomised block execution to inhibit control-flow graph reconstruction by dynamic analysis, and variable (data) splitting to obfuscate data structures. In Figure 1 we show the obfuscation of a simple iterative Fibonacci function as an example of what we mean by obfuscation. We can observe the increase of program variables, a flattened code structure, here depicted in a sequence of tests, driven by a dispatcher whose block selector is determined by complex numerical expressions which are computed at each program block, and weird numbers used to hide internal data-flow and values. While all these techniques can be combined together to protect the code from several models of attack, it is worth noting that each obfuscation strategy is designed to protect the code from one particular kind of attack. However, as most of these techniques are empirical, the major challenges in code protecting transformations are: (1) the design of provably correct code transformations that do not inject flaws when protecting code, and (2) the ability to prove that a certain obfuscation strategy is more effective than another w.r.t. some given attack model.
In this paper we consider a quite general model of attack, propose a measure to compare different obfuscations and define a best obfuscation strategy.

1.2. The challenge

There are several ways in which formal methods, like model checking and abstract interpretation, can be useful for an attacker. For example they can serve to discover software vulnerabilities to be exploited for inject malicious code or to defeat software protection (see e.g. [27, 30, 15]). Some recent approaches aims at the automatic deobfuscation and reverse engineer of protected code [34, 33, 13, 21, 20].
More in general, the aim of any attack is to disclose some program property. Due to the undecidability of generic program analysis and impossibility of VBB obfuscation, measuring the potency of a code transformation defeating WB attacks means specifying precisely the perimeter of the possible attack model. We focus on program properties expressible as formulas in $\forall$CTL*, because they cover a wide range of program properties the attacker may want to disclose. For example, it is known that many data-flow analyses can be cast to model checking of safety formulas. Indeed, computing the results of a set of data-flow equations is equivalent to computing a set of states that satisfies a given modal/temporal logic specification [28, 29]. Even if several interesting properties are not directly expressed as safety properties, because they are existentially quantified over paths, their complements are relevant as well and are indeed safety properties, i.e. they are requested to hold for all reachable states. As we explain later, abstraction techniques (like abstract interpretation and abstract model checking), are necessary for the attacker to cope with the complexity of program analysis to reduce the time and the resources that need to be invested in disclosing the property.
As a concrete example of sensible program properties, suppose the attacker is interested in disclosing the *live* variables of a program. Roughly speaking, a variable x is *live* at a program point iff there exists a

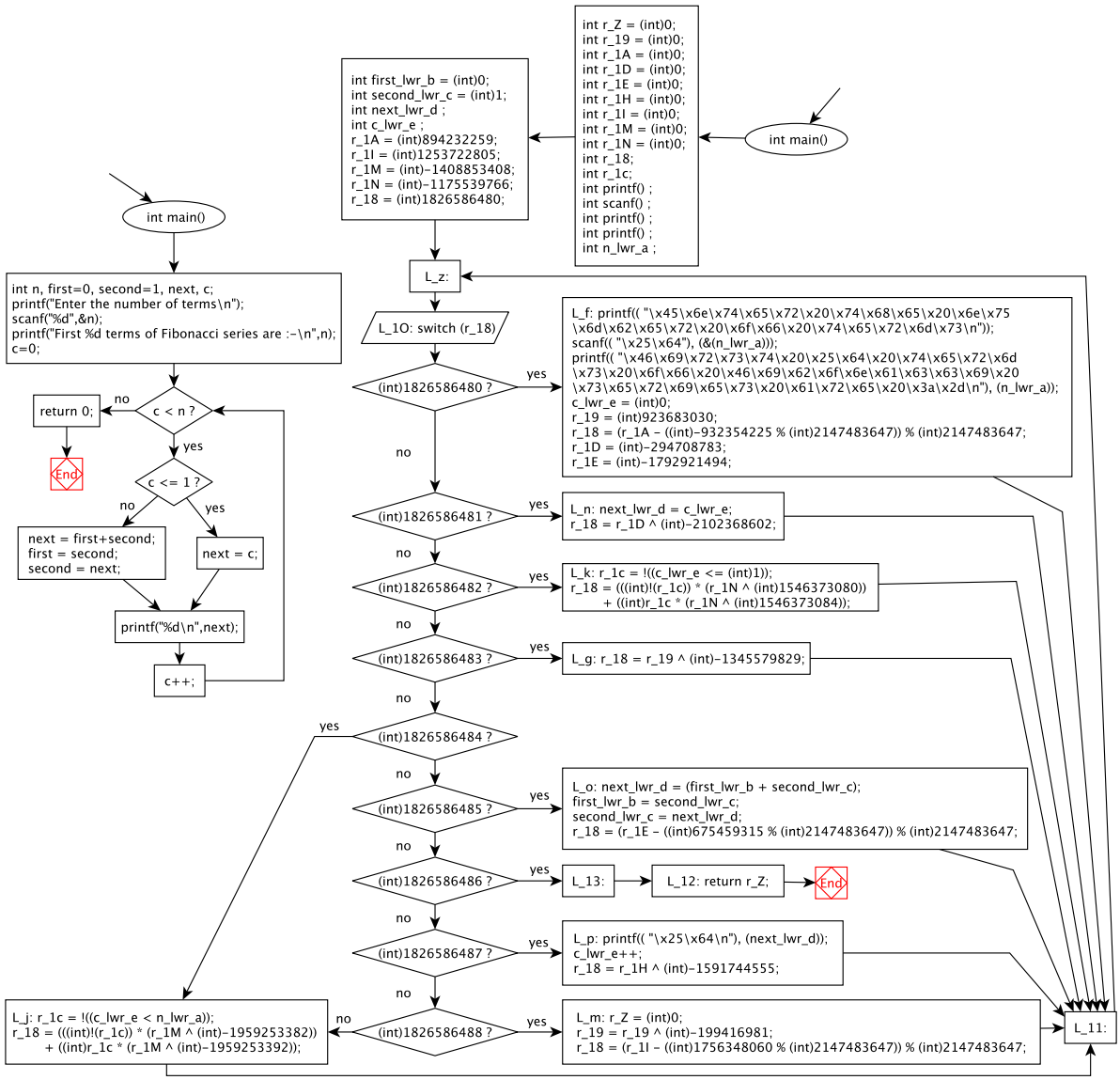


Fig. 1. Source and obfuscated Fibonacci code

path that uses its value before it is modified. We recall that x is *modified* if it appears in the left-hand side of an assignment and it is *used* if it appears in a guard or in the right-hand side of an assignment to a variable $y \neq x$. Knowing that a variable is non-live can allow to discard large portions of dead code and thus to simplify the program analysis. The study of liveness of a variable x can be done by model checking a stronger property in ∀CTL* that formalises the following: at least an instruction using the value of x should be executed between any two different modifications of x. As we explain later this property is called in the literature *very busy expression*. As a simple example, consider the following program written in pseudocode taken from [29].

```
1: while (even(x))  {
2:     x = x div 2;
3: }   y = 2;
```

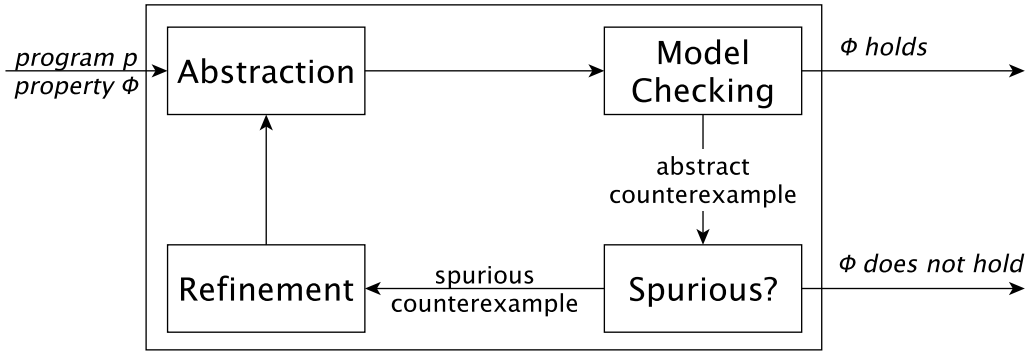


Fig. 2. Counterexample guided abstraction refinement (CEGAR) framework

Note that x satisfies the previous property because: 1) if the values stored in x is odd it is never modified; and 2) if it is even, then x is modified within the body of the loop but then it is used in the guard of the loop. Formally, this can be proved by model-checking an abstraction of the concrete Kripke structure, where the values of x are partitioned in two classes: even values and odd values. This leads to a simple abstract model with just four states (whose size is independent from the initial value of x and its range of values). We show how this program can be obfuscated to make the analysis as hardest as possible in Section 5: proving the same property on the obfuscated program will require to consider an abstract model whose number of states grows linearly with the size of the domain of x.
In this context, program analysis is therefore the model checking of a $\forall$CTL* formula on a possibly abstract model associated with the program. The complexity of software analysis requires automated methods and tools for WB attack to code. Since the attacker aims to disclose the property within a bounded time and using bounded resources, approximate methods such as abstract interpretation [10] or abstract model checking [5] are useful to cope with the complexity of the problem. Abstraction allows to consider sets of values (instead of all individual values separately) for the same variable, thus reducing the overall number of states in the model, while keeping only the relevant information which is necessary for the analysis. Safety properties expressed in $\forall$CTL* can be model-checked using abstraction refinement techniques (CEGAR [4]) as in Fig. 2. An initial (over-approximated) abstraction of the program is model-checked against the property ϕ. If the verification proves that ϕ holds, then it is disclosed. Similarly, if an abstract counterexample is found that corresponds to a concrete counterexample, it is disclosed that ϕ is not valid. An abstract counterexample that is present in the existential over-approximation but not in the concrete model is called *spurious*. If a spurious counterexample is found the abstraction is refined to eliminate it and the verification is repeated on the refined abstraction. Of course, the coarser the abstraction that can be used to verify the property is, the more effective the attack is. Indeed, the worst case for the attacker is when the verification cycle must be repeated until the refined abstraction coincides with the concrete model.

1.3. Contribution

We propose a systematic model deformation that induces a systematic transformation on the program (obfuscation). The idea is to transform the source program in such a way that:

1. its semantics is preserved: the model of the original program is isomorphic to the (reachable) part of the model of the obfuscated program (Theorem 4.1);
2. the performance is preserved;
3. the property ϕ that one wants to hide is preserved by the transformation (Theorem 4.2);
4. such transformation forces the CEGAR framework to ultimately refine the abstract model of the transformed program into the concrete model of the original program (Theorem 4.3). Therefore any abstraction-based model checking on the obfuscated program becomes totally ineffective.

CEGAR can be viewed as a learning procedure, devoted to learn the partition (abstraction) which provides

a (bisimulation) state equivalence. Our transformation makes this procedure extremely inefficient. Note that several instances of the CEGAR framework are possible depending on the chosen abstraction and refinement techniques (e.g. predicate refinement, partition refinement) and that CEGAR can be used in synergy with other techniques for compact representation of the state space (e.g., BDD) and for approximating the best refined abstraction (e.g., SAT solvers). Notably, CEGAR is employed in state-of-the-art tools as Microsoft's SLAM engine for the SDV framework [23] and, more in general, in automatic software verification tools like the open-source software verifier CPAchecker [22]. Here we focus on the original formulation of CEGAR based on partition refinement, but we believe that our technique can be extended to all the other settings. By understanding the model structures that make CEGAR inefficient we provide a better understanding of the limitations of abstraction refinement.
As many obfuscating transformations, our method relies on the concept of *opaque expressions* and *opaque predicate* that are expressions whose value is difficult for the attacker to figure out. Opaque predicates and expressions are formulas whose value is uniquely determined (i.e. a constant), independently from the parameters they receive, but this is not immediately evident from the way in which the formula is written. For example it can be proved that the formula $x^2 - 34y^2 \neq 1$ is always true for any integer values of x and y. Analogously, the formula $(x^2 + x)$ **mod** $2 \neq 0$ is always false. These expression/predicate are, in general, constructed using number properties that are hard for an adversary to evaluate. Of course such predicates can be parameterised so that each instance will look slightly different. Opaque predicates/expressions are often used to inject dead code in the obfuscated program in such a way that program analysis cannot just discard it. For example, if the guard $x^2 - 34y^2 \neq 1$ is used in a conditional statement, then the program analysis should consider both the "then" and the "else" branches, while only the first is actually executable. In this paper: i) opaque expressions will be used to hide from the attacker the initial values of the new variables introduced by our obfuscation procedure; and ii) opaque predicates will be used to add some form of nondeterminism originated from model deformations. The effects of the opaque expressions and predicates will be similar: since the attacker will not be able to figure out their actual values, all the possible values have to be taken into account.

Plan of the paper. In Section 2 we recall CEGAR and fix the notation. In Section 3 we introduce the concept of model deformation and define the measure of obfuscation w.r.t. a given property. In Section 4 we define a best obfuscation strategy and state the main results. In Section 5 we show how to apply our approach to obfuscate flow analysis. Some concluding remarks are in Section 6
This article is the full and revised version of the article published in [3]. More precisely, the new contributions of this paper with respect to [3] are:

- the entire step-by-step application of our obfuscation technique to the running example;
- the application of our approach to the class of data-flow analyses (see the introduction and Section 5);
- the inclusion of extended definitions and explanations to make the paper self-contained;
- the inclusion of all proofs of main results.

Related works. With respect to previous approaches to code obfuscation, all intended to defeat specific abstractions viewed as code attacks, our main contribution is to define the first transformation that defeats the refinement strategy, making our approach independent on the specific attack carried out by abstract model checking. The use of model-driven reasoning gives a clear advantage over existing methods. By exploiting the abstraction refinement algorithms we can make the effort of refining an attack hard in an optimal way.
Most existing works dealing with practical code obfuscation are motivated by either empirical evaluation or by showing how specific models of attack are defeated, e.g., decompilation, program analysis, tracing, debugging (see [9] for a comprehensive survey of most known methods for making code secure). Along these lines, [32] firstly considered the problem of defeating specific and well identified attacks, in this case control-flow structures. More recently [1] shows how suitable transformations may defeat symbolic execution attacks. We follow a similar approach in defeating abstract model-checking attacks by making abstraction refinements maximally inefficient. The advantage in our case is in the fact that we consider abstraction refinements as targets of our code protecting transformations. This allows us both to extract suitable metrics and to apply our transformations to *all* model checking-based attacks.
A first attempt to formalise in a unique framework a variety of models of attack has been done in [16, 12, 17] in terms of abstract interpretation. The idea is that, given an attack implemented as an abstract interpreter, a transformation is derived that makes the corresponding abstract interpreter incomplete, namely returning

the highest possible number of false alarms. The use of abstract interpretation has the advantage of making it possible to include in the attack model the whole variety of program analysis tools. While this approach provides methods for understanding and comparing qualitatively existing obfuscations with respect to specific attacks defined as abstract interpreters, none of these approaches considers transformations that defeat the abstraction refinement, namely the procedure that allows to improve the attack towards a full disclosure of the obfuscated program properties.
Even if the relation between false alarms and spurious counterexamples is known [18] to the best of our knowledge, no obfuscation methods have been developed in the context of formal verification by abstract model checking, or more in general by exploiting structural properties of computational models and their logic.

2. Setting The Context

2.1. Temporal logic and Abstract Model Checking

We consider the fragment ∀CTL* of the branching time temporal logic CTL* [6, 14]. The formulas in ∀CTL* do not contain existential quantifiers and the universal properties are expressed through the path quantifier ∀ ("for all futures") that quantifies over (infinite) execution sequences. The temporal operators G (**G**enerally, always), F (**F**inally, sometime) X (ne**X**t time), and U (**U**ntil) express properties of a single execution sequence. These operators, as well as other syntactic possibilities, can be freely nested in a formula. Given a set *Prop* of propositions, ranged by $p, q, ...$, the set *Lit* of *literals* ℓ is defined as

$$Lit = Prop \cup \{\neg q \mid q \in Prop\} \cup \{\mathsf{true}, \mathsf{false}\}.$$

State formulas ϕ and *Path formulas* ψ are inductively defined by the following grammar, where $\ell \in Lit$ (observe that negation is present only at the level of literals):

$$\text{State formulas:} \quad \phi ::= \ell \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall\psi$$

$$\text{Path formulas:} \quad \psi ::= \phi \mid \psi \wedge \psi \mid \psi \vee \psi \mid G\psi \mid F\psi \mid X\psi \mid U(\psi, \psi)$$

Any state partition $P \subseteq \wp(\Sigma)$ defines an abstraction merging states into abstract states, i.e., an *abstract state* is a set of concrete states and the abstraction function $\alpha_P : \Sigma \to \wp(\Sigma)$ maps each state s into the partition class $\alpha_P(s) \in P$ that contains s. The abstraction function can be lifted to a pair of adjoint functions $\alpha_P : \wp(\Sigma) \to \wp(\Sigma)$ and $\gamma_P : \wp(\Sigma) \to \wp(\Sigma)$, such that for any $X \in \wp(\Sigma)$, $\alpha_P(X) = \bigcup_{x \in X} P(x)$ [25]. When the partition P is clear from the context we omit the subscript. A partition P with abstraction function α induces an *abstract Kripke structure* $\mathcal{K}_P = \langle \Sigma^\sharp, R^\sharp, I^\sharp, \| \cdot \|^\sharp \rangle$ that has abstract states in $\Sigma^\sharp = P$, ranged by $s^\sharp$, and is defined as the existential abstraction induced by P:

- $R^\sharp(s_1^\sharp, s_2^\sharp)$ iff $\exists s, t \in \Sigma.\ R(s,t) \wedge \alpha(s) = s_1^\sharp \wedge \alpha(t) = s_2^\sharp$,
- $s^\sharp \in I^\sharp$ iff $\exists t \in I.\ \alpha(t) = s^\sharp$,
- $\| \mathsf{p} \|^\sharp \stackrel{\text{def}}{=} \{\ s^\sharp \in \Sigma^\sharp \mid\ s^\sharp \subseteq \| \mathsf{p} \|\ \}$,

An abstract path in the abstract Kripke structure $\mathcal{K}_P$ is denoted by $\pi^\sharp = \{s_i^\sharp\}_{i \in \mathbb{N}}$. The abstract path associated with the concrete path $\pi = \{s_i\}_{i \in \mathbb{N}}$ is the sequence $\alpha(\pi) = \{\alpha(s_i)\}_{i \in \mathbb{N}}$. Vice versa, we denote by $\gamma(\pi^\sharp)$ the set of concrete paths whose abstract path is $\pi^\sharp$, i.e., $\gamma(\pi^\sharp) = \{\ \pi \mid\ \alpha(\pi) = \pi^\sharp\ \}$. A counterexample to the formula ϕ is either a finite abstract path or an infinite one represented as a finite abstract path followed by a loop. Abstract model checking is sound by construction: *If there is a concrete counterexample for ϕ then there is also an abstract counterexample for it*. Spurious counterexamples may happen: *If there is an abstract counterexample for ϕ then there may or may not be a concrete counterexample for ϕ.*

2.2. Counter-Example Guided Abstraction Refinement

With an abstract Kripke structure $\mathcal{K}^\sharp$ and a formula ϕ, the CEGAR algorithm works as follows [4]. $\mathcal{K}^\sharp$ is model checked against the formula. If no counterexample to $\mathcal{K}^\sharp \models \phi$ is found, the formula ϕ is satisfied and we

conclude. If a counterexample $\pi^\sharp$ is found which is not spurious, i.e., $\gamma(\pi^\sharp) \neq \emptyset$, then we have an underlying concrete counterexample and we conclude that ϕ is not satisfied. If the counterexample is spurious, i.e., $\gamma(\pi^\sharp) = \emptyset$, then $\mathcal{K}^\sharp$ is further refined and the procedure is repeated. The procedure illustrated in Fig. 2 induces an abstract model-checker attacker that can be specified as follows in pseudocode.

```
Input: program p, property φ
P = init(p, φ);
K = kripke(P, p);
c = check(K, φ);
while (c != null && spurious(p, c)) {
    P = refine(K, p, c);
    K = kripke(P, p);
    c = check(K, φ); }
return ((c == null), P);
```

Here we denote by $init(\cdot)$ a function that takes a program p and the property ϕ and returns an initial abstraction P (a partition of variable domains); a function $kripke(\cdot)$ that generates the abstract Kripke structure associated with a program p and the partition P; a function $check(\cdot)$ that takes an abstract Kripke structure K and a property ϕ and returns either **null**, if $K \models \phi$, or a (deterministically chosen) counterexample c; a predicate *spurious* that takes the program p and an abstract counterexample c and returns true if c is a spurious counterexample and false otherwise; and a function $refine(K, p, c)$ that returns a partition refinement so to eliminate the spurious counterexample c. As the model is finite, the number of partitions that refine the initial partition is also finite and the algorithm terminates by returning a pair: a boolean that states the validity of the formula, and the final partition that allows to prove it.
If several spurious counterexamples exist, then the selection of one instead of another may influence the refinements that are performed. For example, the same refinement that eliminates a spurious counterexample may cause the disappearance of several other ones. However, all the spurious counterexamples must be eliminated. When we assume that $check(\cdot)$ is deterministic, we just fix a total order on the way counterexamples are found. For example, we may assume that a total order on states is given (e.g., lexicographic) and extend it to paths.
Central in CEGAR is partition refinement. The algorithm identifies the shortest prefix $\{s_i^\#\}_{i\in[0,k+1]}$ of the abstract counterexample $\pi^\sharp$ that does not correspond to a concrete path in the model. The second to last abstract state $s_k^\#$ in the prefix, called a *failure state*, is further partitioned by refining the equivalence classes in such a way that the spurious counterexample is removed. To refine $s_k^\#$, the algorithm classifies the concrete states $s \in s_k^\#$ in three classes:

- *Dead states:* they are reachable states $s \in s_k^\#$ along the spurious counterexample prefix but they have no outgoing transitions to the next states in the spurious counterexample, i.e., there is some concrete path prefix $\pi \in \gamma(\{s_i^\#\}_{i\in[0,k]})$ such that $s \in \pi$ and for any $s' \in s_{k+1}^\#$ it holds $\neg R(s, s')$.
- *Bad states:* they are non-reachable states $s \in s_k^\#$ along the spurious counterexample prefix but have outgoing transitions that cause the spurious counterexample, i.e., for any concrete path prefix $\pi \in \gamma(\{s_i^\#\}_{i\in[0,k]})$ we have $s \notin \pi$, but $R(s, s')$ for some concrete state $s' \in s_{k+1}^\#$.
- *Irrelevant states:* they are neither dead nor bad, i.e., they are not reachable and have no outgoing transitions to the next states in the counterexample.

Example 2.1 (Dead, bad and irrelevant states). Consider the abstract path prefix $\{s_0^\#, s_1^\#, s_2^\#, s_3^\#\}$ in Fig. 3. Each abstract state is represented as a set of concrete states (the smaller squares). The arrows are the transitions of the concrete Kripke structure and they induce abstract transitions in the obvious way. We use a thicker borderline to mark $s_0^\#$ as an initial abstract state and a dashed borderline to mark $s_2^\#$ as a failure state. The only dead state in $s_2^\#$ is r, because it can be reached via a concrete path from an initial state but there is no outgoing transition to a state in $s_3^\#$. The states s and t are bad, because they are not reachable from initial states, but have outgoing transitions to states in $s_3^\#$. The states u and w are irrelevant.

CEGAR looks for the coarsest partition that separates bad states from dead states. The partition is obtained

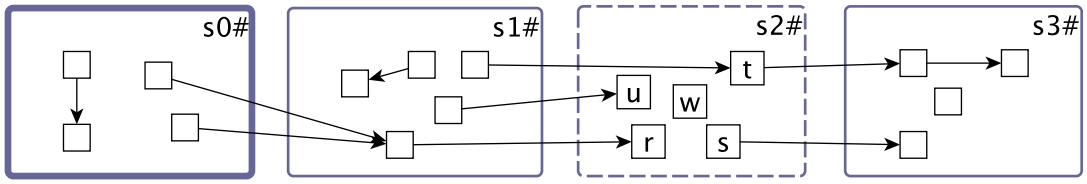


Fig. 3. Dead, bad and irrelevant states

by refining the partition associated with each variable. The chosen refinement is not local to the failure state, but it applies globally to all states: It defines a new abstract Kripke structure for which the spurious counterexample is no longer valid. Finding the coarsest partition corresponds to keeping the size of the new abstract Kripke structure as small as possible. This is known to be a NP-hard problem [4], due to the combinatorial explosion in the number of ways in which irrelevant states can be combined with dead or bad states. In practice, CEGAR applies a heuristic: irrelevant states are not combined with dead states. The opposite option of separating bad states from both dead and irrelevant states is also viable.
In the following we assume that states in Σ are defined as assignments of values to a finite set of variables $x_1, ..., x_n$ that can take values in finite domains $D_1, ..., D_n$. Partitions over states are induced by partitions on domains. A *partition* P of variables $x_1, ..., x_n$ is a function that sends each x_i to a partition $P_i = P(x_i) \subseteq \wp(D_i)$. Given the abstractions associated with partitions $P_1, ..., P_n$ of the domains $D_1, ..., D_n$, the states of the abstract Kripke structure are defined by the possible ranges of values that are assigned to each variable according to the corresponding partition.

2.3. Programs

We let $\mathcal{P}$ be the set of programs written in the syntax of guarded commands [7] (e.g., in the style of CSP, Occam, XC), according to the grammar below:

$$\begin{array}{llll} d & ::= & x \in D \mid d, d & \qquad g ::= x \in V \mid \mathit{true} \mid g \wedge g \mid g \vee g \mid \neg g \\ a & ::= & x = e \mid a, a & \qquad c ::= g \Rightarrow a \mid c|c \end{array} \qquad p ::= (d; g; c)$$

where x is a variable, $V \subseteq \bigcup_i D_i$ is a finite set of values, and e is a well-defined expression over variables. A *basic declaration* $x \in D$ assigns a domain D to the variable x. A *declaration* is a non-empty list of basic declarations. We assume that all the variables appearing in a declaration d are distinct. A *basic guard* is a membership predicate $x \in V$ or true. A *guard* g is a formula of propositional logic over basic guards. We write $x \notin V$ as a shorthand for $\neg(x \in V)$. An *action* is a non-empty list of assignments. A single assignment $x = e$ evaluates the expression e in the current state and updates x accordingly. If multiple assignments $x_1 = e_1, ..., x_k = e_k$ are present, the expressions $e_1, ..., e_k$ are evaluated in the current state and their values are assigned to the respective variables. We require that all the variables appearing in the left-hand side of multiple assignments are distinct. As a consequence, the order of assignments is not relevant. A *basic command* consists of a guarded command $g \Rightarrow a$: it checks if the guard g is satisfied by the current state, in which case it executes the action a to update the state. Commands can be composed in parallel: any guarded command whose guard is satisfied by the current state can be applied. A *program* $(d; g; c)$ consists of a declaration d, an initialisation proposition g and a command c, where all the variables in g and c are declared in d.

Example 2.2 (A sample program). We consider the following running example program (in pseudocode) that computes in y the square of the value initially stored in x.

```
1: y = 0;
2: while (x > 0) {
3:     y = y + 2*x - 1;
4:     x = x - 1;
5: } output(y);
```

For simplicity we assume the possible values assigned to variables are in quite limited ranges, but starting

with larger sets of values would not change the outcome of the application of the CEGAR algorithm (in particular the size of the abstract Kripke structure where the formula ϕ defined in Example 2.5 is satisfied would not change). Let d be the declaration

$$d \stackrel{\text{def}}{=} x \in \{0,1,2\}, y \in \{0,1,2,3,4,5\}, pc \in \{1,2,3,4,5\}$$

The translation of the previous program in the syntax of guarded commands is the program

$$p = (d \ ; \ g \ ; \ c_1 | c_{2a} | c_{2b} | c_3 | c_4).$$

Intuitively, it is obtained by adding an explicit variable pc for the program-counter and then encoding each line of the source code as a basic command. We write it below using CSP-like syntax to help the reading.

```
def x in {0,1,2} , y in {0,1,2,3,4,5} , pc in {1,2,3,4,5};  % d
init pc = 1;                                                 % g
do pc in {1}                  => pc=2, y=0                   % c1  \
[] pc in {2} /\ x notin {0} => pc=3                          % c2a |
[] pc in {2} /\ x in {0}    => pc=5                          % c2b > c
[] pc in {3}                  => pc=4, y=y+(2*x)-1           % c3  |
[] pc in {4}                  => pc=2, x=x-1                 % c4  /
od
```

Note that when $pc = 5$ the program stops because no guarded command is applicable.
In this context, an attacker may want to check if y is ever assigned the value 2, which can be expressed as the property: $\phi \stackrel{\text{def}}{=} \forall\, G\ (pc \in \{1\} \vee y \notin \{2\})$ (i.e. for all paths, for all states in the path it is never the case that $pc \neq 1$ and $y = 2$).

Let $d = (x_1 \in D_1, ..., x_n \in D_n)$. A state $s = (x_1 = v_1, ..., x_n = v_n)$ of the program $(d; g; c)$ is an assignment of values to all variables in d, such that for all $i \in [1, n]$ we have $v_i \in D_i$ and we write $s(x)$ for the value assigned to x in s. Given $c = (g_1 \Rightarrow a_1 | \cdots | g_k \Rightarrow a_k)$, we write $s \models g_j$ if the guard g_j holds in s and $s[a_j]$ for the state obtained by updating s with the assignment a_j.

Example 2.3 (A sample execution). Suppose we take the initial state $s_0 = (x = 1, y = 1, pc = 1)$. The only command whose guard is satisfied by s_0 is c_1 ($s_0 \models pc \in \{1\}$). The execution of c_1 updates the state to $s_1 = s_0[pc = 2, y = 0] = (x = 1, y = 0, pc = 2)$. The only command whose guard is satisfied by s_1 is c_{2a}, whose execution updates the state to $s_2 = s_1[pc = 3] = (x = 1, y = 0, pc = 3)$. Then, the guard of c_3 holds in s_2, leading to $s_3 = (x = 1, y = 1, pc = 4)$. In s_3 the guard of c_4 holds, leading to $s_4 = (x = 0, y = 1, pc = 2)$. Finally, in s_4 the guard of c_{2b} holds, leading to $s_5 = (x = 0, y = 1, pc = 5)$, where the program stops because none of the guards holds in s_5.

The concrete Kripke structure $\mathcal{K}(p) = \langle \Sigma, R, I, \| \cdot \| \rangle$ associated with $p = (d; g; c)$ is defined as follows: the set of states Σ is the set of all states of the program; the set of transitions R is the set of all and only transitions (s, s') such that there is a guarded command $g_j \Rightarrow a_j$ in c with $s \models g_j$ and $s' = s[a_j]$; the set of initial states I is the set of all and only states that satisfy the guard g; the set of propositions is the set of all sentences of the form $x_i \in V$ where $i \in [1, n]$ and $V \subseteq D_i$; the interpretation function is such that $\| x_i \in V \| = \{s \mid s(x_i) \in V\}$.

Example 2.4 (A concrete Kripke structure). The Kripke structure associated with the program p from Example 2.2 has 90 states, one for each possible combination of the values assigned to its variables x, y, pc. The initial states are those where $pc = 1$ (they are 18 in total).

Example 2.5 (A step of CEGAR). Take again the program p and the property ϕ the attacker wants to disclose from Example 2.2.
According to the CEGAR strategy, the attacker can start with the coarsest abstraction that partitions the state according to the subformulas of ϕ and to the control flow conditions in the program. For the sake of the example, here we choose to start with an even coarser abstraction than the above, this is always an option (see [7]). Therefore, assume the attacker starts with the following initial partition:

$$x : \{\{0\}, \{1,2\}\} \qquad y : \{\{2\}, \{0,1,3,4,5\}\} \qquad pc : \{\{1\}, \{2,3,4,5\}\} \tag{1}$$

The corresponding abstract Kripke structure has just 8 states (see Fig. 4) with 4 initial states marked with

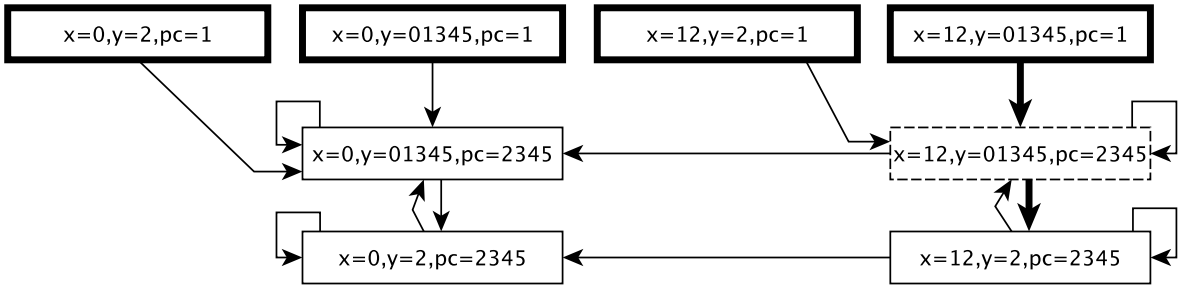


Fig. 4. An abstract Kripke structure

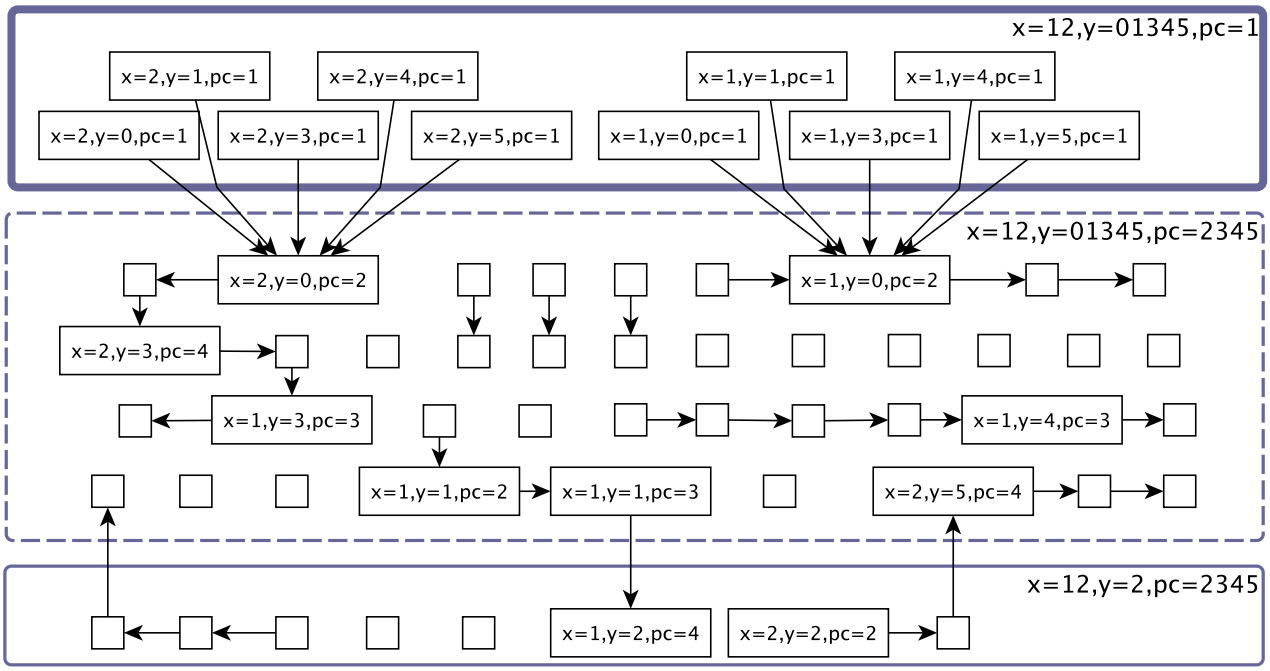


Fig. 5. Failure state

bold borderline in Fig. 4, where, to improve readability, we write, e.g., $y = 01345$ instead of the more verbose $y \in \{0, 1, 3, 4, 5\}$.
There are several paths that lead to counterexamples for ϕ. One such path is the one marked with bold arrows in Fig. 4. It is detailed in Fig. 5 by showing the underlying concrete states. It is a spurious counterexample, because there is no underlying concrete path. The abstract failure state is $(x \in \{1, 2\}, y \in \{0, 1, 3, 4, 5\}, pc \in \{2, 3, 4, 5\})$, depicted with dashed borderline in Fig. 4. It contains one bad concrete state $(x = 1, y = 1, pc = 3)$, two dead states $((x = 1, y = 0, pc = 2)$ and $(x = 2, y = 0, pc = 2))$ and 37 irrelevant states (see Fig. 5, where the underlying concrete states of the abstract failure state $(x \in \{1, 2\}, y \in \{0, 1, 3, 4, 5\}, pc \in \{2, 3, 4, 5\})$ are shown).
By applying the CEGAR refinement we get the following refined partition (remember that irrelevant states can be merged with bad ones but not with dead ones):

$$x : \{\{0\}, \{1, 2\}\} \qquad y : \{\{2\}, \{0\}, \{1, 3, 4, 5\}\} \qquad pc : \{\{1\}, \{2\}, \{3, 4, 5\}\} \tag{2}$$

Thus the corresponding abstract Kripke structure has now 18 states 6 of which are initial states. While the

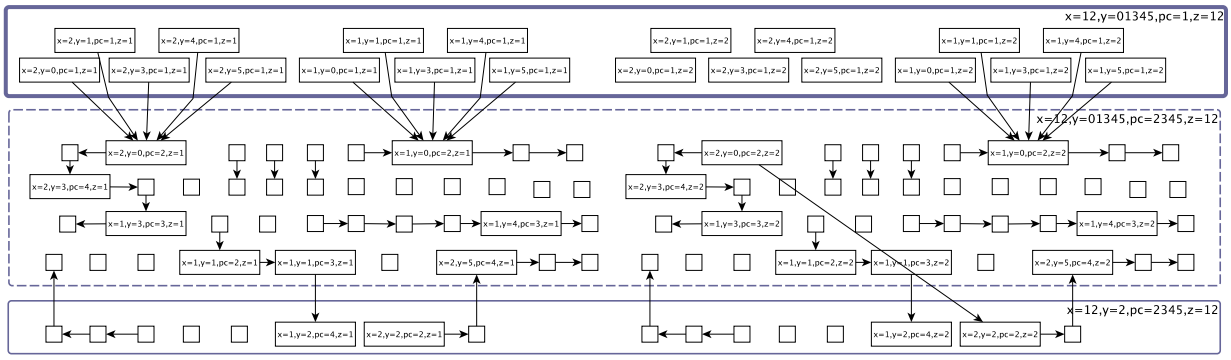


Fig. 6. A detail of a deformation

previously considered spurious counterexample has been removed, another one can be found and, therefore, the CEGAR refinement must be repeated, (see Fig. 7 discussed in Example 3.1 for further steps).

3. Model Deformations

We introduce a systematic model deformation making CEGAR hard. The idea is to transform the Kripke structure, by adding states and transitions in a conservative way. We want to somehow preserve the semantics of the model, while making the abstract model checking based on the CEGAR refinement strategy less efficient, in the sense that only trivial (identical) partitions can be used to prove the property. In other words, any non-trivial abstraction induces at least one spurious counterexample.
Let $\mathbb{M}$ be the domain of all models specified as Kripke structures. Formally, a *model deformation* is a mapping between Kripke structures $\mathfrak{D} : \mathbb{M} \to \mathbb{M}$ such that for a given formula ϕ and $\mathcal{K} \in \mathbb{M}$: $\mathcal{K} \models \phi \;\Rightarrow\; \mathfrak{D}(\mathcal{K}) \models \phi$ and there exists a partition P such that $\mathcal{K}_P \models \phi \;\Rightarrow\; \mathfrak{D}(\mathcal{K}_P) \not\models \phi$. In this case we say that $\mathfrak{D}$ is a deformation for the partition P. Thus, a model deformation makes abstract model checking imprecise yet keeping the validity of the given formula.
Moreover, we show that the deformation of the Kripke structures we consider are induced by transformations of the source program, that act as an obfuscation strategy against an attack specified by an abstract model-checker + CEGAR. Accordingly, we say that an *obfuscation* is a transformation $\mathcal{O} : \mathcal{P} \to \mathcal{P}$ such that for a given formula ϕ and program $p \in \mathcal{P}$: $\mathcal{K}(p) \models \phi \;\Rightarrow\; \mathcal{K}(\mathcal{O}(p)) \models \phi$ and there exists a partition P such that $\mathcal{K}(p)_P \models \phi \;\Rightarrow\; \mathcal{K}(\mathcal{O}(p))_P \not\models \phi$.

3.1. An Example

Consider the program p from Example 2.2. The first step of refinement with the CEGAR algorithm with the initial partition (1) (described in Example 2.5) results in the partition (2). Intuitively, a deformation of the Kripke structure that forced the CEGAR algorithm to split the sets of variable values in classes smaller than the ones in partition (2) would weaken the power of CEGAR.
To this aim, consider a deformation $\mathfrak{D}(\mathcal{K})$ of the concrete Kripke structure $\mathcal{K}$ of Example 2.4 obtained by duplicating $\mathcal{K}$ in such a way that one copy is kept isomorphic to the original one, while the second copy is modified by adding and removing some transitions to make the CEGAR algorithm less efficient. The copies can be obtained by introducing a new variable $z \in \{1, 2\}$: for $z = 1$ we preserve all transitions, while for $z = 2$ we change them to force a finer partition when a step of the CEGAR algorithm is applied. For example, in the replica for $z = 2$, let us transform the copy of the dead state $(x = 2, y = 0, pc = 2)$ into a bad state. This is obtained by adding and removing some transitions. After this transformation, assuming an initial partition analogous to partition (1),

$$x : \{\{0\}, \{1, 2\}\} \quad y : \{\{2\}, \{0, 1, 3, 4, 5\}\} \quad pc : \{\{1\}, \{2, 3, 4, 5\}\} \quad z : \{\{1, 2\}\}$$

where all the values of the new variable z are kept together, we obtain an abstract Kripke structure isomorphic to the one of Fig. 4, with the same counterexamples. However, when we focus on the spurious counterexample, the situation is slightly changed. This is shown in Fig. 6, where the relevant point is the overall shape of the model and not the actual identity of each node. Roughly it combines two copies of the states in Fig. 5: those with $z = 1$ are on the left and those with $z = 2$ are on the right. The abstract state $(x \in \{1, 2\}, y \in \{0, 1, 3, 4, 5\}, pc \in \{2, 3, 4, 5\}, z \in \{1, 2\})$ is still a failure state, but it has three bad states and three dead states:

bad states	dead states
$(\ x=1\ ,\ y=1\ ,\ pc=3\ ,\ z=1\ )$	$(\ x=1\ ,\ y=0\ ,\ pc=2\ ,\ z=1\ )$
$(\ x=1\ ,\ y=1\ ,\ pc=3\ ,\ z=2\ )$	$(\ x=1\ ,\ y=0\ ,\ pc=2\ ,\ z=2\ )$
$(\ x=2\ ,\ y=0\ ,\ pc=2\ ,\ z=2\ )$	$(\ x=2\ ,\ y=0\ ,\ pc=2\ ,\ z=1\ )$

The bad state $(x = 2, y = 0, pc = 2, z = 2)$ and the dead states $(x = 1, y = 0, pc = 2, z = 2)$ and $(x = 1, y = 4, pc = 3, z = 1)$ are incompatible. Therefore, the refinement leads to the partition

$$x : \{\{0\}, \{1\}, \{2\}\} \qquad z : \{\{1\}, \{2\}\} \qquad y : \{\{2\}, \{0\}, \{1, 3, 4, 5\}\} \qquad pc : \{\{1\}, \{2\}, \{3, 4, 5\}\}$$

where all values of x are separated. In Section 4 we show how this deformation is derived from a systematic obfuscation of the program.

3.2. Measuring Obfuscations

Intuitively, the larger the size of the abstract Kripke structure to be model checked without spurious counterexamples is, the harder is for the attacker to reach its goal. The interesting case is of course when the property ϕ holds, but the abstraction used by the attacker leads to spurious counterexamples.
We propose to measure and compare obfuscations on the basis of the size of the abstract Kripke structure where the property can be directly proved. Since the attacker has limited resources, a good obfuscation is the one that forces the attacker to model check a Kripke structure as large as possible, thus diminishing the usefulness of the abstraction. Computing the size of the more abstract Kripke structure that allows to prove the property is theoretically sound but impractical because finding it is an NP-hard problem. This is exactly the reason why CEGAR applies some heuristic to approximate the best abstraction. Therefore we propose to exploit CEGAR to give a bound that can be effectively computed. Note that, in the same spirit as CEGAR, here the number of refinements is less relevant than the size of the abstract Kripke structure that the attacker has to finally check.
As the abstract states are generated by a partition of the domains of each variable, the size is obtained just as the product of the number of partition classes for each variable. As obfuscations can introduce any number of additional variables over arbitrary domains, we consider only the size induced by the variables in the original program (otherwise increasing the number of variables could increase the measure of obfuscation without necessarily making CEGAR ineffective).
In the following we assume that p is a program with variables X and variables of interest $Y \subseteq X$ and ϕ is the formula that the attacker wants to prove.

Definition 3.1 (Size of a partition). Given a partition P of X, we define the *size* of P w.r.t. Y as the natural number $\prod_{y \in Y} |P(y)|$.

Definition 3.2 (Measure of obfuscation). Let $\mathcal{O}(p)$ be an obfuscated program. The *measure* of $\mathcal{O}(p)$ w.r.t. ϕ and Y, written $\#_\phi^Y \mathcal{O}(p)$, is the size of the final partition P w.r.t. Y as computed by the above model of the attacker.

Our definition is parametric w.r.t. to the heuristics implemented in *check*() (choice of the counterexample) and *refine*() (how to partition irrelevant states).
There are several reasons why the above measure is more significant than other choices, like taking into account the number (and the size) of refinements traversed by CEGAR. The main one is that our measure is independent from the order in which spurious counterexamples are eliminated. In fact, the refinement used to eliminate one particular spurious counterexample can also eliminate other ones and diminish the number of necessary refinements. Having to consider the permutations of the spurious counterexamples would make the

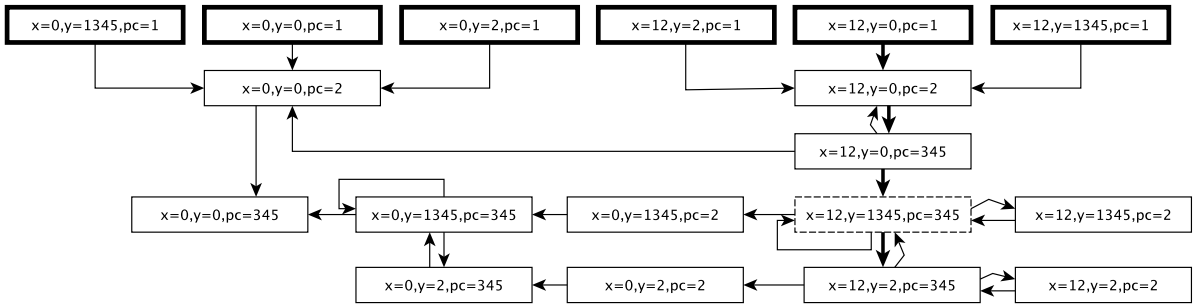


Fig. 7. A refined abstract Kripke structure

measure impractical. Moreover, it allows for the definition of an optimal obfuscation, because it is bounded by the size of the concrete Kripke structure. This would not be the case if the number of refinements were accounted for.

Definition 3.3 (Comparing obfuscations). Given a program p and a formula ϕ and a set of variables Y, the obfuscation $\mathcal{O}_1(p)$ is as good as $\mathcal{O}_2(p)$, written $\mathcal{O}_1(p) \geq_\phi^Y \mathcal{O}_2(p)$, if $\#_\phi^Y \mathcal{O}_1(p) \geq \#_\phi^Y \mathcal{O}_2(p)$.

It follows that the best measure associated with an obfuscation is $\prod_{y \in Y} |D_y|$, where D_y denotes the domain of y. This is the case where the abstraction separates all the concrete values that the variables in Y may assume. Note that any obfuscation whose measure is $\prod_{y \in Y} |D_y|$ makes any abstraction useless to prove the property.

Example 3.1 (Ctd.). Let us consider again the running example and compare it with the semantically equivalent program p' below:

```
def x in {0,1,2},  y in {0,1,2,3,4,5}, pc in {1,2};
init pc = 1;
do pc in {1} => pc=2, y=x*x
od
```

Let x, y be the variables of interest. We have $\#_\phi^{\{x,y\}} p' = 2$, because the initial partition (1) is sufficient to prove that the property ϕ holds.
For the obfuscated program p, the size of the initial partition is just 4 (see partition (1) and the corresponding abstract Kripke structure in Fig. 4), and after one step of the CEGAR refinement the size of the computed partition is 6 (see partition (2) and the corresponding Kripke structure in Fig. 7). Since spurious counterexamples are still present, one more step of refinement is needed. When the attacker executes the procedure on the failure state marked with dashed border in Fig. 7, the result is the partition

$$x : \{\{0\},\{1\},\{2\}\} \qquad y : \{\{0\},\{1\},\{2\},\{3\},\{4,5\}\} \qquad pc : \{\{1\},\{2\},\{4\},\{3,5\}\}$$

whose size is 15 as it has been necessary to split all values for x and most values for y. Now no more (spurious) counterexample can be found because all the states that invalidate the property are not reachable from initial states. Thus $\#_\phi^{\{x,y\}} p = 15$, while the best obfuscation would have measure 18, which is the product of the sizes of the domains of x and y. As the reader may expect, we conclude that $p \geq_\phi^{\{x,y\}} p'$.
In our running example, for simplicity, we have exploited a very small domain for each variable, but if we take larger domains of values, then $\#_\phi^{\{x,y\}} p'$ and $\#_\phi^{\{x,y\}} p$ remain unchanged, while the measure of the best possible obfuscation would grow considerably. This is because the partition

$$x : \{\{0\},\{1\},\{2,3,...,n\}\} \qquad y : \{\{0\},\{1\},\{2\},\{3\},\{4,5,...,m\}\} \qquad pc : \{\{1\},\{2\},\{4\},\{3,5\}\}$$

offers a valid abstraction to prove the property independently from the upper limits n and m.

In the next section we will show how to automatically generate such a best obfuscation.

4. Best Code Obfuscation Strategy

In the following, given an abstract Kripke structure K and a property ϕ, we let S_ϕ denote the set of abstract states that contain only concrete states that satisfy ϕ, and $S_{\overline{\phi}}$ be the set with at least one concrete state that does not satisfy ϕ. We denote by $\overline{\mathcal{O}}_\phi$ the obfuscation strategy realised by the following algorithm.

```
Input: program p, property φ
 1: P = init(p, φ);
 2: K = kripke(P, p);
 3: (p, K, w, v_w) = fresh(p, K);
 4: (p, K, z, v_z) = fresh(p, K);
 5: S = cover(P);
 6: foreach s# ∈ S {
 // failure path preparation (lines 7-12, Cases 1-2)
 7:     π# = failurepath(s#, K, p, φ);
 8:     while (π# == null) {
 9:         if (not reach(s#, K, p, φ)
10:                 (p, K, v_w) = makereachable(s#, K, p, φ, w, v_w);
11:         else (p, K, v_w) = makefailstate(s#, K, p, φ, w, v_w);
12:         π# = failurepath(s#, K, p, φ); }
//  main cycle (lines 13-19, Case 3)
13:     foreach (x_i, v_1, v_2) ∈ compatible(s#, π#, p) {
14:         (s, t) = pick(x_i, v_1, v_2, s#);
15:         if dead(t, s#, π#, p)
16:                 (p, K, v_z) = dead2bad(t, s#, π#, K, p, z, v_z) ;
17:         else if bad(t, s#, π#, p)
18:                 (p, K, v_z) = bad2dead(t, s#, π#, K, p, z, v_z);
19:         else (p, K, v_z) = irr2dead(t, s#, π#, K, p, z, v_z); }
20: } return p;
```

The algorithm starts by computing an initial partition P and the corresponding abstract Kripke structure K. We want to modify the concrete Kripke structure so that CEGAR will split the abstract states in trivial partition classes for the variables of interest. The idea is to create several replicas of the concrete Kripke structure, such that one copy is preserved while the others will be changed by introducing and deleting transitions. This is obtained by introducing a new variable z over a suitable domain $D_z = \{1, ..., n\}$ such that the concrete Kripke structure is replicated for each value z can take. As a matter of notation, we denote by $(s, z = v)$ the copy of the concrete state s where $z = v$. Without loss of generality, we assume that for $z = 1$ we keep the original concrete Kripke structure. In practice such value of z is hidden by an opaque expression. Actually we use two fresh variables, named w and z (lines 3 and 4): the former is used to introduce spurious counterexamples and failure states in the replica and the latter to force the splitting of failure states into trivial partition classes. The function $fresh(\cdot)$ updates the program p and the Kripke structure K by taking into account the new variables and initialises the variables v_w and v_z that keep track of the last used values for w and z. When a new replica is created, such values are incremented.

The function $cover(\cdot)$ (at line 5) takes the initial partition P and returns a set of abstract states $s_1^\#, ..., s_k^\#$, called a *covering*, such that, together, they cover all non-trivial[1] partition classes of the domains of the variable of interest, i.e. for each variable x_i, with $i \in [1, n]$, for each class $C \in P(x_i)$ such that $|C| > 1$ there is an abstract state $s_j^\#$ with $j \in [1, k]$ and a concrete state $s \in s_j^\#$ such that $s(x_i) \in C$. Note that the set of all abstract states is a (redundant) covering.

For each $s^\# \in \{s_1^\#, ..., s_k^\#\}$ in the covering, there are three possibilities:

1. $s^\#$ does not contain any concrete state that is reachable via a concrete path that traverses only abstract states in S_ϕ;

[1] A partition class is trivial if it contains only one value.

2. $s^\#$ is not a failure state but it contains at least one concrete state that is reachable via a concrete path that traverses only abstract states in S_ϕ;
3. $s^\#$ is already the failure state of a spurious counterexample.

In case (3), $\mathit{failurepath}(s^\#, K, p, \phi)$ (line 7) returns an abstract path that is a counterexample for ϕ, in the other cases the function $\mathit{failurepath}(s^\#, K, p, \phi)$ returns *null* and the algorithm enters a cycle (to be executed at most twice, see lines 8–12) that transforms the Kripke structure and the program to move from cases (1–2) back to case (3), in a way that we will explain later. Once a failure path $\pi^\#$ is found whose failure state is $s^\#$, we consider the partition of concrete states in $s^\#$ in the three classes bad, dead and irrelevant. The goal of lines 13–19 is to create enough replicas so that all concrete states in $s^\#$ will be separated by CEGAR. Whenever two values v_1 and v_2 for the variable x_i would be kept in the same partition we need to force their split. The function $\mathit{compatible}(s^\#, \pi^\#, p)$ returns the set of all such triples and the function $\mathit{pick}(x_i, v_1, v_2, s^\#)$ returns two states s and t in $s^\#$ that differs only for the value of x_i (it must be v_1 in one case and v_2 in the other case). Note that any two such states are either: (i) of the same kind (both bad, both dead, both irrelevant), or (ii) one irrelevant and the other bad. Depending on the kind of t, we create a replica (with a different value for z) using the functions $\mathit{dead2bad}(\cdot)$, $\mathit{bad2dead}(\cdot)$ or $\mathit{irr2dead}(\cdot)$ such that s and t are no more compatible and thus the value v_1 and v_2 for x_i will be split by CEGAR. The detail of the transformations are presented below.

Case (3) (lines 13–19). The core of the algorithm applies to a failure state $s^\#$ of a spurious counterexample $\pi^\#$. In this case the obfuscation method has to guarantee that the CEGAR refinement will split the failure state $s^\#$ by separating all values in the domains of the variables of interest.

Remember that CEGAR classifies the concrete states in $s^\#$ in three classes (bad, dead and irrelevant) and that dead states cannot be merged with bad or irrelevant states. We say that two states that can be merged are *compatible*. The role of the new copies of the Kripke structure is to prevent any merge between concrete states in the set $s^\#$. This is done by making sure that whenever two concrete states $(s, z = 1), (t, z = 1) \in s^\#$ can be merged into the same partition by the CEGAR algorithm, then the states $(s, z = v_z + 1)$ and $(t, z = v_z + 1)$ cannot be merged together, because one is dead and the other is bad or irrelevant.

The function $\mathit{compatible}(s^\#, \pi^\#, p)$ returns the set of triples $(x_i, \{v_1, v_2\})$ such that x_i is a variable of interest and any pair of states $(s, x_i = v_1), (s, x_i = v_2) \in s^\#$ that differ just for the value of x_i are compatible. Thus, the cycle at line 13 considers all such triples to make them incompatible. At line 14, we pick any two compatible states $(s, z = 1)$ and $(t, z = 1)$ such that $s(x_i) = v_1$, $t(x_i) = v_2$ and $s(x) = t(x)$ for any variable $x \neq x_i$. Given the spurious counterexample $\pi^\#$ with failure state $s^\#$ and a concrete state $t \in s^\#$ for the program p, the predicate $\mathit{dead}(\cdot)$ returns true if the state $(t, z = 1)$ is dead (line 15). Similarly, the predicate $\mathit{bad}(\cdot)$ returns true if the state $(t, z = 1)$ is bad (line 17).

If $(t, z = 1)$ is dead (w.r.t. $s^\#$ and $\pi^\#$), then it means that $(s, z = 1)$ is also dead (because they are compatible), so we apply a dead-to-bad transformation to t in the replica for $z = v_z + 1$. This is achieved by invoking the function $\mathit{dead2bad}(\cdot)$ (line 16) to be described below. The transformations $\mathit{bad2dead}(\cdot)$ (line 18) and $\mathit{irr2dead}(\cdot)$ (line 19) apply to the other classifications for $(t, z = 1)$. In more details, in lines 16, 18 and 19 we are concerned with the addition/removal of transitions to replica of the Kripke structure for $z = v_z + 1$ so that a concrete state changes its classification.

In the following, given a concrete state $s = (x_1 = w_1, ..., x_n = w_n)$, we denote by $G(s)$ the guard $x_1 \in \{w_1\} \wedge ... \wedge x_n \in \{w_n\}$ and by $A(s)$ the assignment $x_1 = w_1, ..., x_n = w_n$. Without loss of generality, in the following descriptions of transformations, we assume for brevity that the abstract counterexample is formed by the abstract path prefix $\pi^\# = \{s_0^\#, s_1^\#, s^\#, s_2^\#\}$, with $s^\#$ the failure state and t is the concrete state in $s^\#$ that we want to transform (see Figs. 8–10).

$\mathit{dead2bad}(\cdot)$**.** To make t bad, the function must remove all concrete paths to t along the abstract counterexample $\pi^\#$ and add one transition from t to some concrete state t' in $s_2^\#$. To remove all concrete paths it is enough to remove the transitions from states in $s_1^\#$ to t (see Fig. 8). At the code level, $\mathit{dead2bad}(\cdot)$ modifies each command $g \Rightarrow a$ such that there is some $s' \in s_1^\#$ with $s' \models g$ and $t = s'[a]$. Given t and $s_1^\#$, let $S(g \Rightarrow a) = \{s' \in s_1^\# \mid s' \models g \wedge t = s[a]\}$. Each command $c = (g \Rightarrow a)$ such that $S(c) \neq \emptyset$ is changed to the

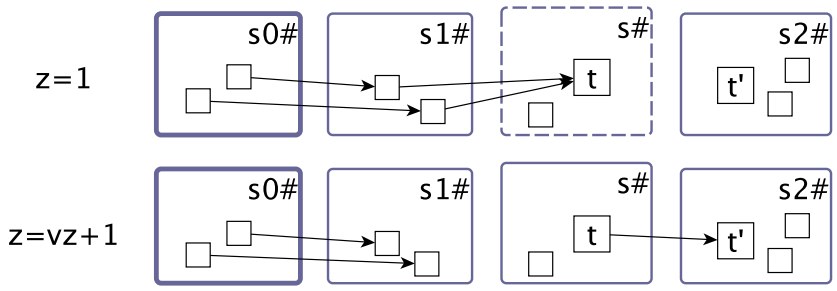


Fig. 8. From dead to bad

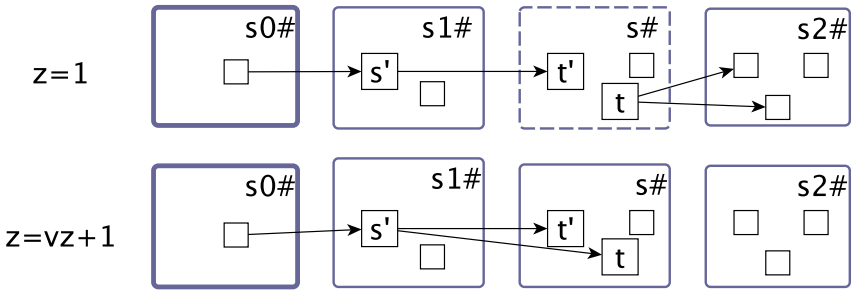


Fig. 9. From bad to dead

command

$$g \wedge \left(z \notin \{v_z + 1\} \vee \bigwedge_{s' \in S(c)} \neg G(s')\right) \Rightarrow a.$$

When $z \neq v_z + 1$ the updated command is applicable whenever c was applicable. When $z = v_z + 1$ the command is not applicable to the states in $S(c)$. To add the transition from t to a state t' in $s_2^\#$ it is enough to add the command $z \in \{v_z + 1\} \wedge G(t) \Rightarrow A(t')$.

bad2dead$(\cdot)$**.** The function selects a dead state t' in the failure state $s^\#$ and a concrete path $\pi = s_0, s', t'$ to t' along the abstract counterexample $\pi^\#$. To make t dead in the replica with $z = v_z + 1$, the function adds a concrete transition from s' to t and removes all transitions leaving from t to concrete states in $s_2^\#$ (see Fig. 9). To insert the transition from s' to t, the command $G(s') \wedge z \in \{v_z + 1\} \Rightarrow A(t)$ is added. To remove all transitions leaving from t to concrete states in $s_2^\#$, the function changes the guard g of each command $g \Rightarrow a$ such that $t \models g$ and $t[a] \in s_2^\#$ to $g \wedge (z \notin \{v_z + 1\} \vee \neg G(t))$, which is applicable to all the states different from t where $g \Rightarrow a$ was applicable as well as to the replicas of t for $z \neq v_z + 1$.

irr2dead$(\cdot)$**.** To make t dead, the function builds a concrete path to t along the abstract counterexample $\pi^\#$. As before, it selects a dead state t' in the failure state and a concrete path $\pi = s_0, s', t'$ to s along the abstract counterexample $s_0^\#, s_1^\#, s^\#$. To make t dead the function adds a transition from s' (i.e., the state that immediately precedes t' in π) to t (see Fig. 10). For the program it is sufficient to add a new command with guard $G(s') \wedge z \in \{v_z + 1\}$ and whose assignment is $A(t)$.

From cases (1–2) to case (3) (lines 7–12). The predicate $reach(s^\#, K, p, \phi)$ is true if we are in case (2) and false if we are in case (1). In both cases we apply some preliminary transformations to K and p after which $s^\#$ is brought to case (3). The functions *makereachable*$(\cdot)$ and *makefailstate*$(\cdot)$ perform such transformations by returning a modified programs and its corresponding modified Kripke structure.[2]

[2] At the code level, addition and removal of transitions is realised as detailed before.

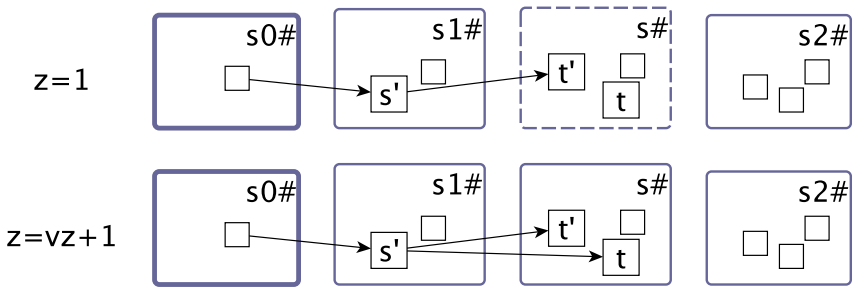


Fig. 10. From irrelevant to dead

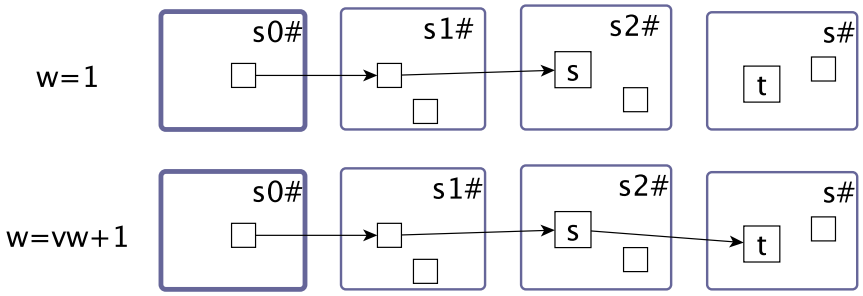


Fig. 11. Case 1

In more details, the function $makereachable(s^{\#}, K, p, \phi, w, v_w)$ transforms the Kripke structure so that in the end $s^{\#}$ contains at least one concrete state that is reachable via a concrete path that traverses only abstract states in S_ϕ, moving from case (1) to cases (2) or (3), while the function $makefailstate(s^{\#}, K, p, \phi, w, v_w)$ takes $s^{\#}$ satisfying case (2) and it returns a modified Kripke structure where $s^{\#}$ now falls in case (3). Before applying $makereachable(\cdot)$, $s^{\#}$ does not contain any concrete state that is reachable via a concrete path that traverses only abstract states in S_ϕ. The function $makereachable(\cdot)$ selects a concrete path π such that its abstract counterpart $\alpha(\pi)$ goes trough states in S_ϕ. Let s be the last concrete state of π. For $w = v_w + 1$ it selects a concrete state $t \in s^{\#}$ and adds a transition from s to t (see Fig. 11, where $\alpha(\pi) = (s_0^{\#}, s_1^{\#}, s_2^{\#})$ and $s_0^{\#}, s_1^{\#}, s_2^{\#}, s^{\#} \in S_\phi$). If $s^{\#}$ is a failure state we are back to case (3), otherwise we move to case (2). To transform $s^{\#}$ into a failure state, $makefailstate(\cdot)$ adds a transition from a non-reachable state $t \neq s$ in $s^{\#}$ to some concrete state t' such that $\alpha(t') \in S_{\overline{\phi}}$ (see Fig. 12, where $s_0^{\#}, s_1^{\#}, s^{\#} \in S_\phi$ and $s_2^{\#} \in S_{\overline{\phi}}$). We note that the selected state t must be non-reachable before the transformation: if t has some incoming transitions, they must be deleted. Now $s^{\#}$ is a failure state and we are back to case (3).

4.1. Main Results

We are now ready to state the main results of the paper.
First, we want to guarantee that the semantics of the program is preserved by the obfuscation. Intuitively, this is because all the obfuscations we have discussed maintain the original Kripke structure associated with a distinguished value of the new variables w and z that are introduced. Indeed, we prove that when the obfuscated program is executed with initial values $w = 1$ and $z = 1$ it is guaranteed to behave exactly as the original program: if the attacker were given the extra power to guess the right values for z and w, then the reachable part of the deformed Kripke structure would be isomorphic to the Kripke structure of the original program. This can be prevented, e.g., by exploiting opaque expressions to initialise the variables w and z, thus hiding the right values to use from the attacker: since the attacker cannot disclose the value returned

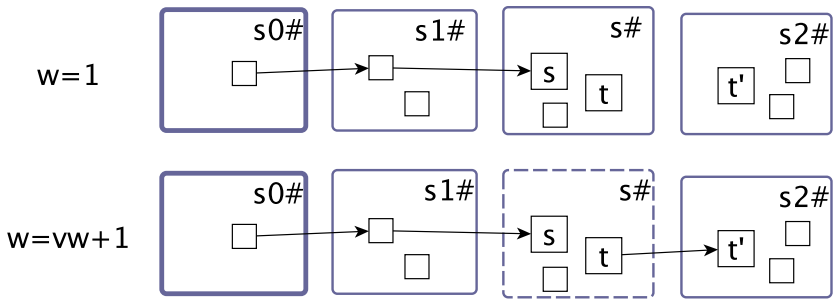


Fig. 12. Case 2

by opaque expressions, he has to take into account all possible values for w and z and thus run CEGAR on the deformed Kripke structure.

Theorem 4.1 (Embedding). For $p = (d; g; c)$ and its obfuscated version $\overline{\mathcal{O}}_\phi(p) = ((d, z \in D_z); g; c')$, then $\mathcal{K}(p)$ is isomorphic to $\mathcal{K}((d, w \in \{1\}, z \in \{1\}); g; c')$.

Proof. Let w and z be the fresh variables. By construction, when we introduce a new value for w or z and add a command $g \Rightarrow a$, then there is a distinguished value v_w of w (respectively, v_z of z) for which the guard g is falsified in any state where $w = 1$ (respectively $z = 1$). Moreover, when we modify a command $c = (g \Rightarrow a)$ to $c' = (g' \Rightarrow a)$, the assignment is not changed and for any state s where $w = 1$ (respectively $z = 1$) we have $s \models g$ iff $s \models g'$. In more detail:

- dead to bad transformation:
  - for each command $c = (g \Rightarrow a)$ that is changed to the command
    $$g \wedge \left(z \notin \{v_z + 1\} \vee \textstyle\bigwedge_{s' \in S(c)} \neg G(s')\right) \Rightarrow a.$$
    we immediately have that when $z = 1$ the predicate $z \notin \{v_z + 1\}$ is true and the guard $g \wedge \left(z \notin \{v_z + 1\} \vee \bigwedge_{s' \in S(c)} \neg G(s')\right)$ is logically equivalent to g.
  - the added command $z \in \{v_z + 1\} \wedge G(t) \Rightarrow A(t')$ is clearly not applicable when $z = 1$.
- bad to dead transformation:
  - the added command $G(s') \wedge z \in \{v_z + 1\} \Rightarrow A(t)$ is clearly not applicable when $z = 1$.
  - for each command $g \Rightarrow a$ that is changed into $g \wedge (z \notin \{v_z + 1\} \vee \neg G(t)) \Rightarrow a$, we immediately have that when $z = 1$ the predicate $z \notin \{v_z + 1\}$ is true and the guard $g \wedge (z \notin \{v_z + 1\} \vee \neg G(t))$ is logically equivalent to g.
- irrelevant to dead transformation: the guard $G(s') \wedge z \in \{v_z + 1\}$ of the added command $G(s') \wedge z \in \{v_z + 1\} \Rightarrow A(t)$ does not apply to states where $z = 1$.

For the cases (1) and (2), the required transformations are analogous to the ones discussed above. □

The isomorphism at the level of Kripke structures guarantees that the obfuscation does not affect the number of steps required by any computation, i.e., to some extent the efficiency of the original program is also preserved.
Second, the obfuscation preserves the property ϕ of interest when the program is executed with any input data for w and z, i.e. ϕ is valid in all replicas.

Theorem 4.2 (Soundness). $\mathcal{K}(p) \models \phi$ iff $\mathcal{K}(\overline{\mathcal{O}}_\phi(p)) \models \phi$.

Proof. If $\mathcal{K}(p) \not\models \phi$, then a counterexample can be found in the original Kripke structure that by Theorem 4.1 is present also in $\mathcal{K}(\overline{\mathcal{O}}_\phi(p))$. If $\mathcal{K}(p) \models \phi$, then by construction, only spurious counterexamples are added. To see this, let us consider the different transformations. When a bad or irrelevant state s is transformed to

a dead state, it is made reachable, but we remove any outgoing transition that leads to a state where the property is violated. When a dead state s is transformed to a bad state, we do add a transition to a state where the property is violated, but we make s not reachable. □

It is worth to mention that the isomorphism at the level of Kripke structures from Theorem 4.1 guarantees that the semantics is preserved entirely, i.e. not only ϕ is preserved in all replicas (Theorem 4.2) but any other property is preserved when the obfuscated program is run with $w \in \{1\}$ and $z \in \{1\}$.
The next result guarantees the optimality of obfuscated programs.

Theorem 4.3 (Hardness). $\#_\phi^Y \overline{\mathcal{O}}_\phi(p) = \prod_{y \in Y} |D_y|$.

Proof. Let $X = \{x_1, ..., x_n\}$. By contradiction, let P be the final partition computed by the model-checker attacker and suppose that the size of P w.r.t. X is strictly less than $\prod_{i=1}^{n} |D_{x_i}|$. Then, it means that there is at least one variable x_i and two values $v_1, v_2 \in D_{x_i}$ that are in the same partition class of P. But this is not possible, because for any pair of values v_1, v_2 that x_i can take, by construction of $\overline{\mathcal{O}}_\phi(p)$ there are at least two incompatible states $(t, x_i = v_1)$ and $(t, x_i = v_2)$ that should have been separated by the CEGAR algorithm. This is a consequence of the fact that we have chosen a cover at line 5 of our obfuscation algorithm. □

As a consequence, for any program p and formula ϕ the function mapping $\mathcal{K}(p)$ into $\mathcal{K}(\overline{\mathcal{O}}_\phi(p))$ is a model deformation for all (non-trivial) partitions of the variables of interest.

Theorem 4.4 (Complexity). The complexity of our best code obfuscation strategy is polynomial in the size of the domains of the variables of interest Y.

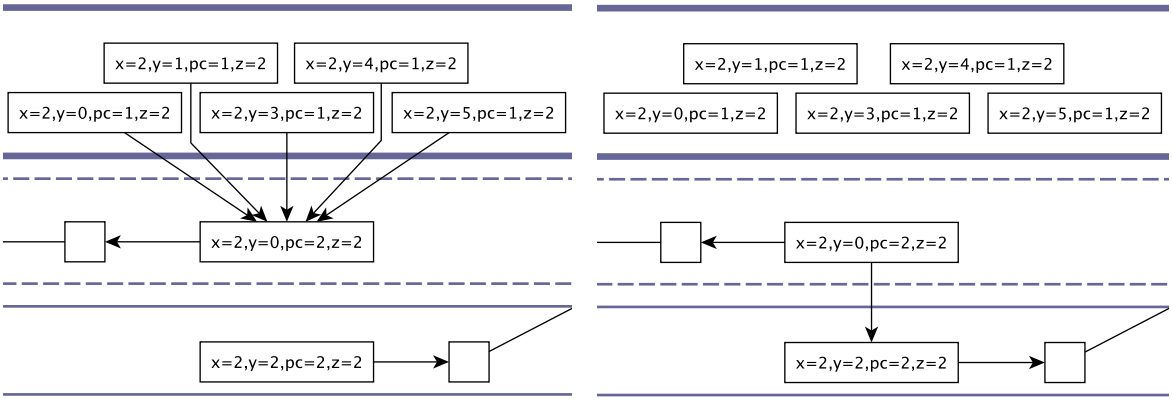
Proof. To estimate the complexity of the best code obfuscation strategy, note that the relevant part of the algorithm is the cycle on lines 12–18 and that the number of required transformations depends linearly in the number of variables of interest and quadratically on the size of their domains. In fact, the number of pairs of values to be distinguished for the variable x is at most $|D_x| \cdot (|D_x| - 1)/2$ and we need one transformation for each such pair. Note however, that in general not all pairs are to be considered and that the same transformation can distinguish more than one pair of values. □

Finally, it is worth noting that the obfuscation that we have discussed assumes that the procedure of CEGAR makes dead states incompatible with both irrelevant and bad states. Even if the attacker followed a more general procedure, where dead states are only incompatible with bad states, our obfuscation strategy can immediately generalised by taking into account this new notion of incompatibility when selecting the pair of values to be separated. Therefore even if the attacker had the power to compute the coarsest partition of the concrete states in the abstract failure state that separates bad states from dead states (which is a NP-hard problem in the presence of irrelevant states) our strategy would force the partition to consist of trivial classes only. In the running example, all we had to do was to apply ten more transformations of irrelevant states.

4.2. Best Obfuscation For The Running Example

We now apply the best code obfuscation strategy to our running example. Consider the abstract Kripke structure in Fig. 4. The failure state $s^\# = (x \in \{1, 2\}, y \in \{0, 1, 3, 4, 5\}, pc \in \{2, 3, 4, 5\})$ covers all the non-trivial partition classes for the variables of interest x and y. Since it is a failure state for an abstract counterexample, we are in case (3). For simplicity, since the transformations for cases (1–2) are not needed, we omit the insertion of the variable w.
The dead state $s = (x = 1, y = 0, pc = 2)$ is incompatible with the irrelevant state $t = (x = 1, y = 1, pc = 2)$, thus the triple $(y, \{1, 2\})$ is incompatible. For the same reason the value 0 for y is also separated from the values $3, 4, 5$. Our obfuscation must separate the values $1, 2$ for x and the values $1, 3, 4, 5$ for y. Therefore at most 6 replicas are needed. In the end, 5 values for z suffices. Let us take the triple $(x, \{1, 2\})$ and let us pick the two dead states $t = (x = 2, y = 0, pc = 2)$ and $s = (x = 1, y = 0, pc = 2)$ in Fig. 5. The algorithm invokes *bad2dead*$(\cdot)$ on state $(t, z = 2)$ to make it incompatible with the dead state $(x = 1, y = 0, pc = 2, z = 2)$. At the code level, we note that all incoming transitions of t are due to the command c_1 (see Fig. 5). To remove them, c_1 becomes $c_{1,1}$.

```
pc in {1} /\ (z notin {2} \/ x notin {2} \/ y notin {2}) => pc=2, y=0     %c11
```



(a) Before the deformation (as in Fig. 5) (b) After the deformation

Fig. 13. A detail of the abstract Kripke structure for z=2

Moreover, to make $(t, z = 2)$ a bad state, is added a transition from the state $(t, z = 2)$, to $(x = 2, y = 2, pc = 2, z = 2)$ with the new command $c_{1,2}$

```
pc in {1} /\ z in {3} /\ x in {1} /\ y in {1}              => pc=3            %c12
```

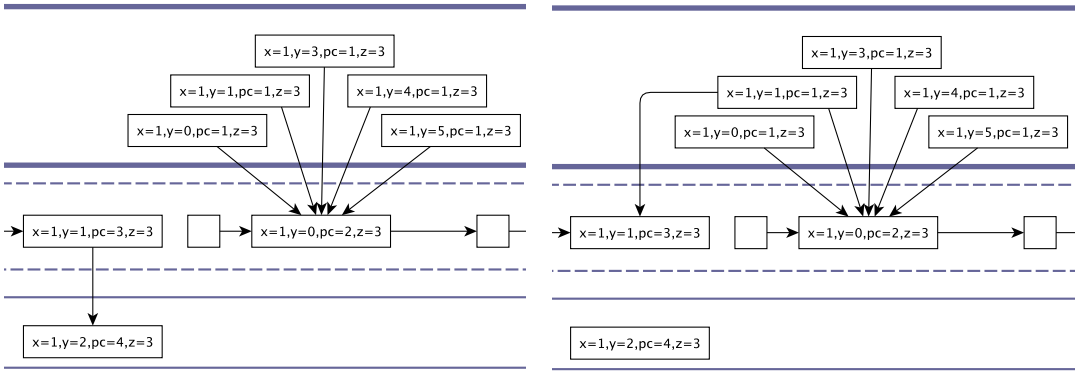
In Fig. 13b we show the relevant changes on the Kripke structure for the replica with $z = 2$ (compare it with Fig. 13a the corresponding detail of Fig. 5).
The other transformations needed consists of: (i) a bad-to-dead transformation for the triple $(y, \{1, 3\})$ (and $z = 3$) that also make incompatible the triples $(y, \{1, 4\})$ and $(y, \{1, 5\})$; (ii) an irrelevant-to-dead transformation for the triple $(y, \{3, 4\})$ (and $z = 4$) that, as a side-effect, also make incompatible the triples $(y, \{4, 5\})$; and (iii) an irrelevant-to-dead transformation for the triple $(y, \{3, 5\})$ (and $z = 5$).
All transformations are described below.
For (i), handling the triple $(y, \{1, 3\})$, let us consider the bad state $t = (x = 1, y = 1, pc = 3)$ and the irrelevant state $s = (x = 1, y = 3, pc = 3)$ in Fig. 5. We apply a bad-to-dead transformation to the state $(t, z = 3)$ to make it incompatible with $(s, z = 3)$. To remove the only outgoing transition from t (see Fig. 5) we must change the guard of the command c_3 to $pc \in \{3\} \wedge (z \notin \{3\} \vee x \notin \{1\} \vee y \notin \{1\})$. This produces the command $c_{3,1}$. Moreover, we need to make $(t, z = 3)$ reachable, e.g., from $s' = (x = 1, y = 1, pc = 1, z = 3)$. Hence we add the command $c_{1,2}$

```
pc in {1} /\ z in {3} /\ x in {1} /\ y in {1}              => pc=3                          %c12
pc in {3} /\ (z notin {3} \/ x notin {1} \/ y notin {1}) => pc=4, y=y+(2*x)-1            %c31
```

Note that as a side effect, the same transformation separates the values $1, 4$ for y and also $1, 5$ for y.
In Fig. 14b we show the relevant changes on the Kripke structure for the replica with $z = 3$ with respect to the original Kripke structure depicted in Fig. 14a.
For (ii), handling the triple $(y, \{3, 4\})$, let us consider the irrelevant states $t = (x = 1, y = 4, pc = 3)$ and $(x = 1, y = 3, pc = 3)$ in Fig. 5. We apply an irrelevant-to-dead transformation to the state $(t, z = 4)$. Since we need to make t reachable, we select, e.g., the state $s' = (x = 1, y = 4, pc = 1)$ and we add a transition towards t. To this aim we add the command $c_{1,3}$.

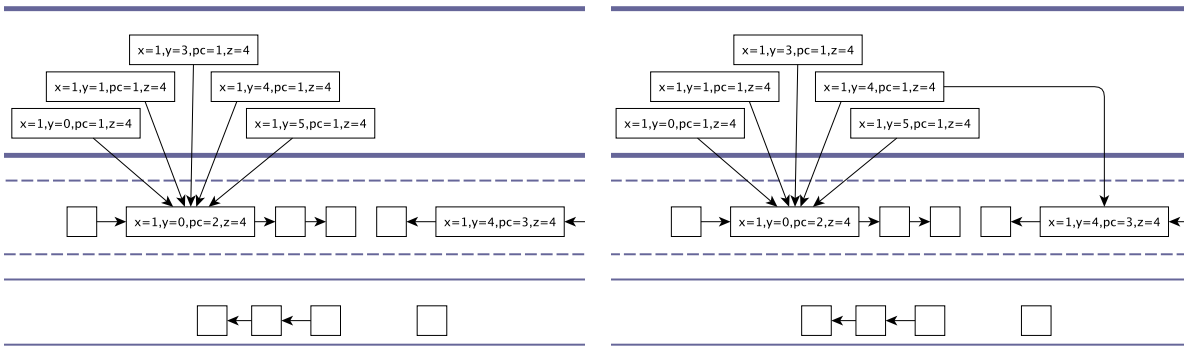
```
pc in {1} /\ z in {4} /\ x in {1} /\ y in {4}      => pc=3                          %c13
```

Note that, as a side effect, the same transformation separates the values $4, 5$ for y. In Fig. 15b we show the relevant changes on the Kripke structure for the replica with $z = 4$ with respect to the original Kripke structure of Fig. 15a.
Finally (iii), it remains to deal with the triple $(y, \{3, 5\})$. To this aim, let us consider the irrelevant states $t = (x = 2, y = 5, pc = 4)$ and $s = (x = 2, y = 3, pc = 4)$ in Fig. 5. We apply an irrelevant-to-dead transformation to the state $(t, z = 5)$. To make t reachable, we select, e.g., the state $s' = (x = 2, y = 5, pc = 1)$ and we add a transition towards t. To this aim we add the command $c_{1,4}$.



(a) Before the deformation (as in Fig. 5) (b) After the deformation

Fig. 14. A detail of the abstract Kripke structure for z=3



(a) Before the deformation (as in Fig. 5) (b) After the deformation

Fig. 15. A detail of the abstract Kripke structure for z=4

```
pc in {1} /\ z in {5} /\ x in {2} /\ y in {5}      => pc=4                    %c14
```

As result of the obfuscation we obtain the program $po = \overline{\mathcal{O}}_\phi(p)$ below:

```
def x in {0,1,2} , y in {0,1,2,3,4,5} , pc in {1,2,3,4,5} , z in {1,2,3,4,5};
init pc = 1;
do pc in {1} /\ (z notin {2} \/ x notin {2} \/ y notin {2}) => pc=2, y=0              %c11
[] pc in {1} /\ z in {3} /\ x in {1} /\ y in {1}            => pc=3                   %c12
[] pc in {1} /\ z in {4} /\ x in {1} /\ y in {4}            => pc=3                   %c13
[] pc in {1} /\ z in {5} /\ x in {2} /\ y in {5}            => pc=4                   %c14
[] pc in {2} /\ x notin {0}                                 => pc=3                   %c2a
[] pc in {2} /\ x in {0}                                    => pc=5                   %c2b
[] pc in {2} /\ x in {2}} /\ y in {0} /\ z in {2}           =>  y=2                   %c21
[] pc in {3} /\ (z notin {3} \/ x notin {1} \/ y notin {1}) => pc=4, y=y+(2*x)-1      %c31
[] pc in {4}                                                => pc=2, x=x-1            %c4
od
```

Let us assume that the attacker starts with the abstraction of the obfuscated program induced by the partition $x : \{\{0\}, \{1, 2\}\}$, $y : \{\{2\}, \{0, 1, 3, 4, 5\}\}$, $pc : \{\{1\}, \{2, 3, 4, 5\}\}$, and $z : \{\{1, 2, 3, 4, 5\}\}$. The abstract Kripke structure is isomorphic to the one in Fig. 4 having several spurious counterexamples for ϕ. One such

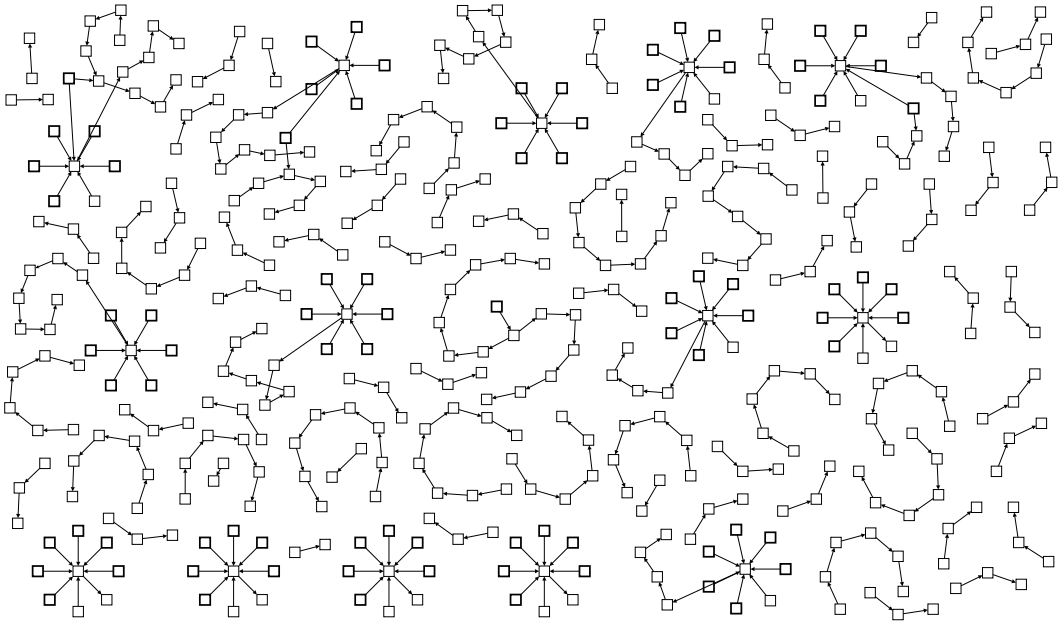


Fig. 16. Kripke structure of the obfuscated program after the first refinement

path is similar to the one in Fig. 4: $\{s_0^\#, s_1^\#, s_2^\#\}$ with:

$$\begin{array}{lcl} s_0^\# & = & x \in \{1,2\}, y \in \{0,1,3,4,5\}, pc \in \{1\}, z \in \{1,2,3,4,5\} \\ s_1^\# & = & x \in \{1,2\}, y \in \{0,1,3,4,5\}, pc \in \{2,3,4,5\}, z \in \{1,2,3,4,5\} \\ s_2^\# & = & x \in \{1,2\}, y \in \{2\}, pc \in \{2,3,4,5\}, z \in \{1,2,3,4,5\}. \end{array}$$

The failure state is $s_1^\#$. It has 5 bad concrete states and 12 dead states. By CEGAR we get the partition

$$x : \{\{0\},\{1\},\{2\}\}\ \ y : \{\{0\},\{1\},\{2\},\{3\},\{4\},\{5\}\}\ \ pc : \{\{1\},\{2\},\{3\},\{4\},\{5\}\}$$

$$z : \{\{1\},\{2\},\{3\},\{4\},\{5\}\}$$

that has only trivial (singletons) classes. Therefore the abstract Kripke structure coincides with the concrete Kripke structure: it has 450 states of which 90 are initial states. Its states are depicted in Fig. 16 where the 90 initial states are marked with thicker lines.
Given that the variables of interest are x and y, the measure of the obfuscation is 18, i.e., it has the maximum value and thus $po \geq_\phi^{\{x,y\}} p \geq_\phi^{\{x,y\}} p'$. We remark that when $z = 1$, po has the same semantics as p and p'.
The guarded command po can be understood as a low-level, flattened description for programs written in any language. In general, the same low-level description can correspond to many different high-level programs. However, it is not difficult to derive, e.g., an ordinary imperative program from a given guarded command. We do so for the reader's convenience.

```
 1:       z = opaque1(x,y,z);
 2: pc1: if ( ( z!=2 || x!=2 || y!=2 ) && opaque2(x,y,z) ) { y=0; goto pc2; }
 3:       else if ( z==3 && x==1 && y==1 ) goto pc3;
 4:       else if ( z==4 && x==1 && y==4 ) goto pc3;
 5:       else if ( z==5 && x==2 && y==5 ) goto pc4;
 6: pc2: if ( z==2 && x==2 && y==0 && opaque3(x,y,z) ) y=2;
 7:       while ( x > 0 ) {
 8: pc3:    if ( z!=3 || x!=1 || y!=1 ) y = y + 2*x - 1;
 9: pc4:    x = x - 1;
10: pc5: } output(y);
```

To hide the real value of z we initialise the variable using an opaque expression $opaque1(x, y, z)$ whose value is 1. Moreover, one has to pay attention to the possible sources of nondeterminism, which can arise when there are two or more guarded commands $g_1 \Rightarrow a_1$ and $g_2 \Rightarrow a_2$ and a state s such that both g_1 and g_2 hold in s. The idea is to introduce opaque predicates in the program so that the exact conditions under which a branch is taken are hard to determine by the attacker, who has to take into account both possibility (*true* and *false*) as a nondeterministic choice. In our example, the sources of nondeterminism are due to the pair of commands $(c_{1,1}, c_{1,2})$, $(c_{1,1}, c_{1,3})$, $(c_{1,1}, c_{1,4})$ and $(c_{2,1}, c_{2a})$. Consequently, we assume two opaque predicates $opaque2(x, y, z)$ and $opaque3(x, y, z)$ are available. In order to preserve the semantics, for $z = 1$ we require that $opaque1(x, y, z)$ returns *true*, while for $opaque2(x, y, z)$ there is no restriction. Finally, since the program counter is an explicit variable in guarded commands, we represent its possible values by labels and use goto instructions accordingly. Thus we write the label *pcn* to denote states where $pc = n$ and write **goto** *pcn* for assignments of the form $pc = n$. Below is the imperative program we were able to derive from the guarded command *po*.

5. Obfuscating Some Flow Analyses

Data flow analyses can be useful to eliminate dead-code and to determine variables that will be bound to constant values at certain program points. An attacker can therefore use data flow techniques to infer program properties that enhance his/her understanding of the program behaviour for a successful reverse engineering and deobfuscation.
In [28, 29] it is shown that data flow analysis can be equivalently expressed in terms of model checking, in the sense that the set of states satisfying a set of data-flow equations is the one that satisfies a corresponding modal/temporal logic specification. In this section we exploit such a correspondence to show that our obfuscation method can be used to prevent attacks aimed at disclosing data flow properties.
To this aim, we point out that our approach can be extended to address program models where transitions are labelled. Simple transition annotations such as *use* and *mod* (indexed by a variable x, an expression e or a definition d) can be used to address data flow properties in program models using modal/temporal logic formulas. They refer, in order, to the fact that the variable x, an expression e, or a definition d has been used in performing the transition or that its value has been modified. At the level of temporal logic we consider parameterised operators w.r.t. transition labels: for A a set of labels, we write $G_A\phi$ to mean that ϕ is valid in all states reachable using transitions whose labels are in A, and $[A]\phi$ to mean that ϕ is valid in all states reachable by any transition whose label is in A.
Below we give some examples of well-studied data flow properties as expressed in ∀CTL*:

- *Constant propagation*: this property refers to the fact that the value of a given expression e does not change in the rest of the computation. This information can be useful in dead code elimination and therefore to reduce the complexity of program analysis by deleting part of the program model, e.g., by removing nondeterminism. The corresponding logic formula is:

$$\mathbf{CP}(e) \stackrel{\mathsf{def}}{=} \forall\mathbf{G}[mod_e]\mathsf{false}$$

  it is satisfied by those states where for any path and for any state along the path there is no outgoing transition that modifies the value of e.
- *Available expression*: this flow analysis aims at detecting if an expression e is available at program point p, i.e. the value of e has not changed since the last time e was computed on the paths to p. The logic formulas for available expression uses inverse modalities. For simplicity, we report here a slightly stronger related property:

$$\mathbf{AE}(e) \stackrel{\mathsf{def}}{=} \forall\mathbf{G}_{\{a|a\neq use_e\}}[mod_e]\mathsf{false}$$

  it is satisfied by those states where for any path and for any state in the path that is reachable via transitions that do not use the value of e, there is no transition that modifies the value of e.
- *Dead variable*: a variable x is dead if its current value is never used. This information can be useful to restrict the set of variables of interest and their values. The corresponding logic formula is:

$$\mathbf{D}(x) \stackrel{\mathsf{def}}{=} \forall\mathbf{G}_{\{a|a\neq mod_x\}}[use_x]\mathsf{false}$$

it is satisfied by those states where for any path and for any state in the path that is reachable via transitions that do not modify the value of x, there is no transition using the value of x.

- *Very busy expression*: this property is a stronger variant than liveness. It means that whenever the expression e is modified it is not possible to reach a state where it is modified again unless it is used first. The corresponding logic formula is:

$$\mathbf{VBE}(e) \stackrel{\mathsf{def}}{=} \forall\mathbf{G}[mod_e]\forall\mathbf{G}_{\{a|a\neq use_e\}}[mod_e]\mathsf{false}$$

it is satisfied by those states where for any path, for any state in the path and for any outgoing transition that modifies the value of e we only reach states where the expression e is available (in the sense explained earlier).

As a concrete example, let us investigate the property $\mathbf{VBE}(x)$ for the program already discussed in the Introduction (taken from [29])

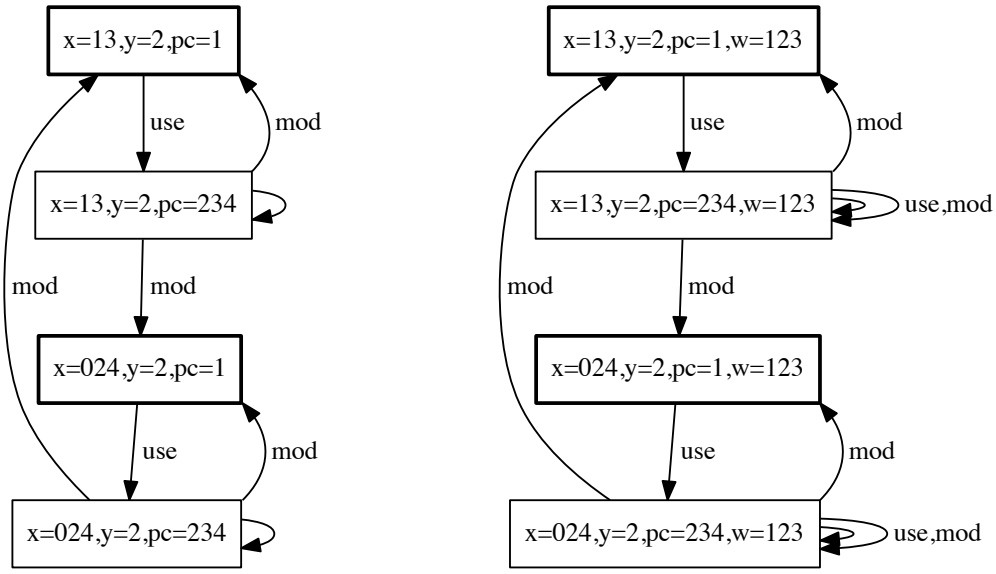
```
1: while (even(x))  {
2:     x = x div 2;
3: } y = 2;
```

and, for simplicity, consider a very small domain for x, say $\{0,1,2,3,4\}$. Using the syntax of guarded commands, the program becomes written as below.

```
def x in {0,1,2,3,4} , y in { 2 } , pc in {1,2,3,4};
init pc = 1;
do pc in {1} /\ even(x) => pc=2
[] pc in {1} /\  odd(x) => pc=3
[] pc in {2}            => x=x div 2, pc=1
[] pc in {3}            => y=2, pc=4
od
```

In order to disclose the property of $\mathbf{VBE}(x)$ an attacker can try to model check an abstract model of the program against the above formula. Using CEGAR, the attacker would start from an abstract model where the domain of x is partitioned in two classes: even values and odd values, and would easily prove that x is a very busy expression. Indeed, in this case, there are no counterexamples to the required property. This can be readily verified by looking at the abstract Kripke structure in Figure 17a.
The application of our best obfuscation strategy leads to the introduction of two spurious counterexamples, according to case (2), with the addition of a variable $w \in \{1,2,3\}$. The corresponding abstract Kripke structure is in Figure 17b, where the two spurious counterexample are evident by observing the newly added self-loops with labels use followed by mod. Then a variable $z \in \{1,2,3,4\}$ is used to separate all values that x can take in the failure states. Omitting the details, we get the obfuscated program:

```
def x in {0,1,2,3,4}, y in {2}, pc in {1,2,3,4}, w in {1,2,3}, z in {1,2,3,4};
init pc = 1;
// original commands are preserved
do pc in {1} /\ even(x) => pc=2
[] pc in {1} /\  odd(x) => pc=3
[] pc in {2}            => x=x div 2, pc=1
[] pc in {3}            => y=2, pc=4
// two new commands to introduce spurious counterexamples
[] w in {2} /\ pc in {3} /\ x in {0}   => x=2, y=2, pc=4
[] w in {3} /\ pc in {2} /\ x in {1}   => x=3, pc=4
// one new command to split (x,2,4)
[] z in {2} /\ pc in {1} /\ x in {2}   => pc=3
// one new command to split (x,1,3)
[] z in {3} /\ pc in {1} /\ x in {3}   => pc=2
// one new command to split (x,0,4)
[] z in {4} /\ pc in {1} /\ x in {2,4} => pc=3
od
```



(a) For the original program

(b) After the introduction of spurious counterexamples (application of case (2) of the obfuscation strategy)

Fig. 17. Abstract Kripke structures

Now, applying CEGAR to the obfuscated program, the property is proved to hold only after having partitioned the domain of x in trivial (singleton) classes.
As before, we can give an imperative program that corresponds to the obfuscated code.

```
 1:       w = opaque1(x,w,z);
 2:       z = opaque2(x,w,z);
 3: pc1: if ( x==2 && z==2 && opaque3(x,w,z) ) goto pc3;
 4:       else if ( x==3 && z==3 && opaque4(x,w,z) ) goto pc2;
 5:       else if ( (x==2 || x==3) && z==4 && opaque5(x,w,z) ) goto pc3;
 6:       while (even(x)) {
 7: pc2:     if ( x==1 && w==3 && opaque6(x,w,z) ) { x=3; goto pc4; }
 8:          x = x div 2;
 9: pc3: } if ( x==0 && w==2 && opaque7(x,w,z) ) { x=2; y=2; goto pc4; }
10:       y = 2;
11: pc4: skip;
```

Of course the value returned by opaque expressions $opaque1(x, w, z)$ and $opaque2(x, w, z)$ must be 1. Since all opaque predicates are in conjunction with values different than 1 for w and z, the semantics of the original program is preserved independently of the results returned by opaque predicates.

6. Discussion

We have shown that it is possible to systematically transform Kripke structures in order to make automated abstraction refinement by CEGAR hard. Addressing refinement procedures instead of specific abstractions makes our approach independent from the chosen abstraction in the attack.

To enforce the protection of the real values of variable w (and analogously for z) initialised by opaque functions against more powerful attacks able to inspect the memory of different program runs, one idea is to use a class of values instead of a single value. This allows the obfuscated code to introduce instructions that assign to w different values in the same class, thus convincing the attacker that the value of w is not invariant.
The complexity of our best code obfuscation strategy is polynomial in the size of the domains of the variables of interest. Moreover, we note that the same algorithm can produce a valuable obfuscation even if one selects a partial cover instead of a complete one: in this case, it is still guaranteed that the refinement strategy will be forced to split all the values appearing in the partial cover. This allows to choose the right trade-off between the complexity of the obfuscation strategy and the measure of the obfuscated program. It is also possible that the algorithm introduces more transformations than strictly necessary. This is because the obfuscation is performed w.r.t. an initial partition. To further reduce the number of transformations, we can apply the obfuscation only at the end of the abstract model checking process, where less pairs of values needs to be separated. Of course, this strategy would not influence the measure of the result.
As already mentioned, our obfuscation assumes that CEGAR makes dead states incompatible with both irrelevant and bad states. Our algorithm can be generalised to the more general setting where dead states are only incompatible with bad states. Therefore even if the attacker had the power to compute the coarsest partition that separates bad states from dead states (which is a NP-hard problem) our strategy would force the partition to consist of trivial classes only.
We can see abstraction refinement as a learning procedure which learns the coarsest state equivalence by model checking a temporal formula. Our results provide a very first attempt to defeat this procedure.
It remains to be investigated how big the text of the best obfuscated program can grow: limiting its size is especially important in the case of embedded systems.
We plan to extend our approach to other abstraction refinements, like predicate refinement and the completeness refinement in [19] for generic abstract interpreters and more in general for a machine learning algorithm. This would make automated reverse engineering hard in more general attack models.

Acknowledgement. We are very grateful to Alberto Lluch-Lafuente for the fruitful discussions we had on the subject of this paper.

References

[1] S. Banescu, C. S. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner. Code obfuscation against symbolic execution attacks. In S. Schwab, W. K. Robertson, and D. Balzarotti, editors, *Proc. 32nd Annual Conference on Computer Security Applications, ACSAC 2016*, pages 189–200. ACM, 2016.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

[3] R. Bruni, R. Giacobazzi, and R. Gori. Code obfuscation against abstract model checking attacks. In I. Dillig and J. Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, volume 10747 of *Lecture Notes in Computer Science*, pages 94–115. Springer, 2018.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[5] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proc. of the 19th ACM Symp. on Principles of Programming Languages (POPL '92)*, pages 343–354. ACM Press, 1992.

[6] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[8] C. Collberg, J. Davidson, R. Giacobazzi, Y. Gu, A. Herzberg, and F. Wang. Toward digital asset protection. *IEEE Intelligent Systems*, 26(6):8–13, 2011.

[9] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, 1977.

[11] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Proc. of the 31st ACM Symp. on Principles of Programming Languages (POPL '04)*, pages 173–185. ACM Press, New York, NY, 2004.

[12] M. Dalla Preda and R. Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.

[13] R. David. *Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes. (Approches formelles de désobfuscation automatique et de rétro-ingénierie de codes protégés)*. PhD thesis, University of Lorraine, Nancy, France, 2017.

[14] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, Mass., 1990.

[15] J. Feist, L. Mounier, and M. Potet. Statically detecting use after free on binary code. *J. Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[16] R. Giacobazzi. Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In *Proc. of the 6th IEEE Int. Conferences on Software Engineering and Formal Methods (SEFM '08)*, pages 7–20. IEEE Press, 2008.

[17] R. Giacobazzi, N. D. Jones, and I. Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'12)*, pages 63–72. ACM Press, 2012.

[18] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In *Proc. of the 8th Int. Static Analysis Symp. (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373. Springer, 2001.

[19] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretation complete. *Journal of the ACM*, 47(2):361–416, March 2000.

[20] J. Kinder. Towards static analysis of virtualization-obfuscated binaries. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, pages 61–70. IEEE Computer Society, 2012.

[21] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In K. Julisch and C. Krügel, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, Second International Conference, DIMVA 2005, Vienna, Austria, July 7-8, 2005, Proceedings*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2005.

[22] S. Löwe. *Effective Approaches to Abstraction Refinement for Automatic Software Verification.* PhD thesis, University of Passau, Germany, 2017.

[23] Microsoft. Static driver verifier website (last consulted november 2017), 2017. `https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier`.

[24] J. Nagra, C. D. Thomborson, and C. Collberg. A functional taxonomy for software watermarking. *Aust. Comput. Sci. Commun.*, 24(1):177–186, 2002.

[25] F. Ranzato and F. Tapparo. Generalized strong preservation by abstract interpretation. *Journal of Logic and Computation*, 17(1):157–197, 2007.

[26] H. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[27] R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pages 156–165. IEEE Computer Society, 2000.

[28] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In D. B. MacQueen and L. Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 38–48. ACM, 1998.

[29] D. A. Schmidt and B. Steffen. Program analysis *as* model checking of abstract interpretations. In G. Levi, editor, *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Springer, 1998.

[30] TCIPG.ORG. Vulnerability assessment tool using model checking, fact sheet (last consulted march 2018), 2018. `https://tcipg.org/sites/default/files/factsheets/FactSheet_Vulnerability-Assessment.pdf`.

[31] R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. In *Proc. 4th Int. Workshop on Information Hiding (IHW '01)*, volume 2137 of *Lecture Notes in Computer Science*, pages 157–168. Springer, 2001.

[32] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson. Protection of software-based survivability mechanisms. In *2001 International Conference on Dependable Systems and Networks (DSN 2001) (formerly: FTCS), 1-4 July 2001, Göteborg, Sweden, Proceedings*, pages 193–202. IEEE Computer Society, 2001.

[33] B. Yadegari. *Automatic Deobfuscation and Reverse Engineering of Obfuscated Code.* PhD thesis, University of Arizona, Tucson, USA, 2016.

[34] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 674–691. IEEE Computer Society, 2015.