Protecting code through Obfuscation

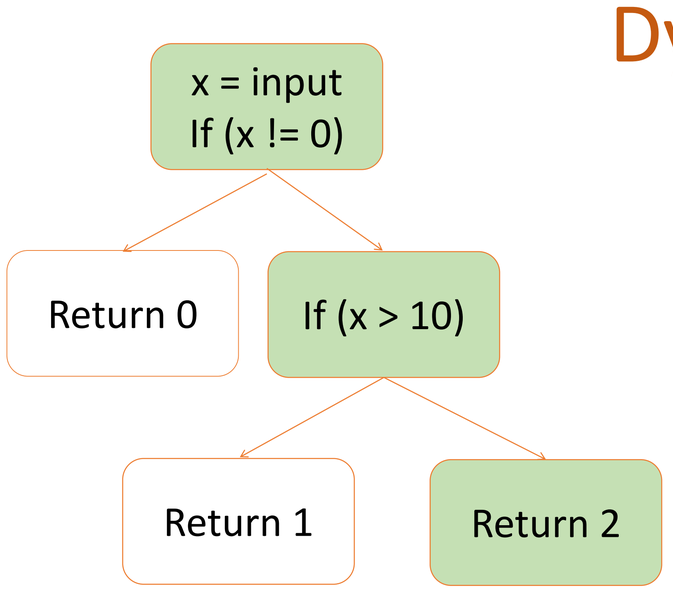
Context

Reverse engineering and code tampering are widely used to extract proprietary assets or bypass security checks from software. **Code protection** techniques try to prevent **man-at-the-end attacks** (*i.e.* when the attacker has full control over the execution environment).
Common obfuscation techniques are quite resilient against basic automatic reverse engineering, however code analysis improves quickly. Attacks based on **Dynamic Symbolic Execution** (DSE) appear to be very efficient.

Challenges

Several dedicated protections against DSE-based attacks have been proposed, yet the state of knowledge is pretty unclear and few techniques have actually been implemented. Moreover DSE has been proven very efficient against **state-of-the-art obfuscation tools** using common transformations such as virtualization and self-modification.
We want to propose a **new class of dedicated** protections making attacks based on DSE inefficient. These techniques should induce a **substantial slowdown** and be **lightweight**.

Dynamic Symbolic Execution



Path constraint: φ = (x≠0) ∧ (x>10)

DSE simulates the execution of a program along its paths, systematically generating inputs for each new discovered branch conditions. Inputs are considered as **symbolic variables** whose values are not fixed. Every time the engine encounters a conditional statement involving an input, it adds a **constraint** to the symbolic value of the input. These constraints are then fed to a **SMT solver** to generate input values leading to new paths.
We can point out three main issues with DSE that can be used to create dedicated protections: *hard constraints*, *path explosion* and *path divergence*.

Hard Constraints

Path constraints can be hard to solve for SMT solvers. For example, specific **non-linear operations** such as multiplication or division substantially increase the complexity. This issue is critical to DSE because if a constraint can not be solved, it reduces the subset of paths the analysis is able to explore.

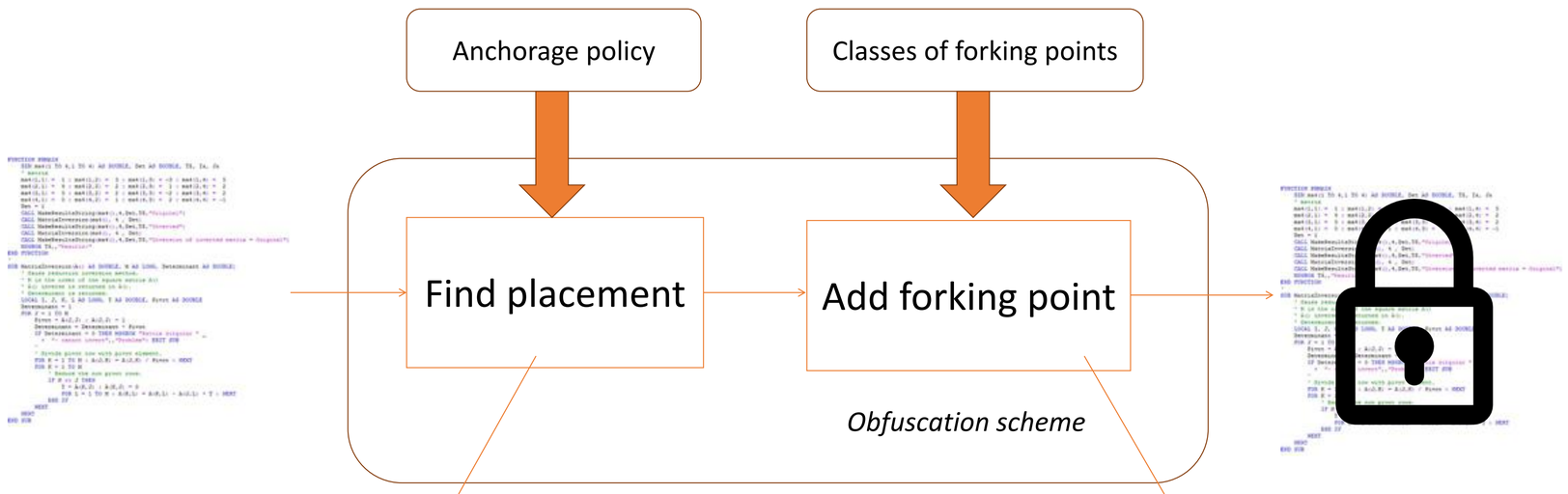
Path Explosion

To explore paths, the symbolic engine needs to **solve constraints** and **store all pending states** in memory. For these two reasons, it is not reasonable to explore a large subset of paths. Thus DSE can realistically explore only a reduced number of paths in a limited amount of time.

Path Divergence

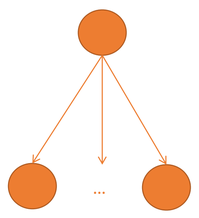
Computing precise and correct path constraints can be difficult for certain programs. If path constraints are not computed precisely it can lead to path divergence: **feasible paths are missed** or **unfeasible paths are taken.** Either way the symbolic analysis is not able to give a correct view of the paths tree.

Obfuscation scheme for **Path-Oriented Protections**



Good placement ?

Independance: forking points do not hinder each others' impact on the symbolic analysis
Optimal composition: every paths contain at least k forking points to ensure a sufficient slowdown



Forking point: Location in code where a path is split into two or more paths

Mathilde OLLIVIER
Directeur : Jean-Yves MARION
Encadrant CEA : Sébastien BARDIN