A Survey of Control-Flow Obfuscations

Anirban Majumdar, Clark Thomborson, and Stephen Drape

Secure Systems Group
Department of Computer Science, The University of Auckland,
Auckland, New Zealand, Private Bag 92019
{anirban,cthombor,stephen}@cs.auckland.ac.nz

Abstract. In this short survey, we provide an overview of obfuscation and then shift our focus to outlining various non-trivial control-flow obfuscation techniques. Along the way, we highlight two transforms having provable security properties: the dispatcher model and opaque predicates. We comment on the strength and weaknesses of these transforms and outline difficulties associated in generating generalised classes of these.

1 Introduction

The motivation for research in obfuscation stems from the problem of software piracy. An obfuscating transform attempts to manipulate code in such a way that it becomes unintelligible to automated program analysis tools used by malicious reverse engineers. It works by performing semantic preserving transformations which aim to increase the difficulty of automatically extracting computational logic out of the code. Depending on the size of software and the complexity of transforms, a human adversary may also find the obfuscated code difficult to comprehend; however, this is not a mandatory requirement.

The first formal definition of obfuscation was given by Collberg et al. [1,2]. They defined an obfuscator in terms of a semantic-preserving transformation function $\mathcal{T}$ which maps a program $\mathcal{P}$ to a program $\mathcal{P}'$ such that if $\mathcal{P}$ fails to terminate or terminates with an error, then $\mathcal{P}'$ may or may not terminate. Otherwise, $\mathcal{P}'$ must terminate and produce the same output as $\mathcal{P}$. Collberg et al. classified obfuscating transforms into three useful categories:

– **Layout obfuscation:** Changes or removes useful information from the intermediate language code or source code, e.g. removing debugging information, comments, and scrambling/renaming identifiers.
– **Data obfuscation:** Targets data and data structures contained in the program, e.g. changing data encoding, variable and array splitting and merging.
– **Control-flow obfuscation:** Alters the flow of control within the code, e.g. reordering statements, methods, loops and hiding the actual control flow behind irrelevant conditional statements.

This paper focuses on the latter category since the first two have been extensively investigated before [1,3].

2 The Dynamic Dispatcher Model

Wang et al. [4] and Chow et al. [5] made the first commendable attempt to lay theoretical foundations for control-flow obfuscations of sequential programs. The theoretical basis in Wang et al.'s technique is the `NP-complete` argument of determining precise indirect branch target addresses of dispatchers in the presence of aliased pointers. Chow et al., on the other hand, uses `PSPACE-complete` argument of determining the reachability of a flattened program dispatcher. Control-flow flattening makes all basic blocks appear to have the same set of predecessors and successors. The actual control-flow during execution is determined dynamically by a dispatcher. Thus, the dispatcher module is the most important component in the flattened (obfuscated) program. In Wang's technique, each flattened basic block while exiting, changes the dispatcher variable through complicated pointer manipulation on some global data structure. Chow et al.'s technique, on the other hand, views the dispatcher as a deterministic finite automaton that determines the overall control-flow of the obfuscated flattened program from a state space of given transitions. However, if the state space is rather small, it will not be difficult to deobfuscate the dispatcher. For this reason, its authors expand the state space by incorporating numerous dummy states.

Since we do not have an average-case hard instance generator for either `NP-complete` or `PSPACE-complete` problems, these theoretical foundations are still incomplete. However even in their current state, these models provide some "fuzzy" confidence in the security of the real-world obfuscation systems they describe. Eventually we would hope to prove security results for some models of obfuscation. Then the only real-world attacks would be to subvert the assumptions of these models, analogous to how a provably-secure cryptographic system can never be "cracked", but may still be subverted e.g. by "social engineering" methods of password discovery.

3 Opaque Predicates

An opaque predicate is a conditional expression whose value is known to the obfuscator, but is difficult for an adversary to deduce statically. A predicate Φ is defined to be *opaque* at a certain program point p if its outcome is only known at obfuscation time. Following Collberg et al. [2], we write Φ_p^F (Φ_p^T) if predicate Φ always evaluates to False (True) at program point p for all runs of the same program. We call such predicates *Opaquely True (False)* at program point p. The opaqueness property is necessary for guaranteeing the resilience of control-flow transformations.

Algebraic predicates have invariants that are based on well-known mathematical axioms [6]. A predicate of this class is $\Phi : [(x(x+1)\%2 == 0]$, which is opaquely true for all integers. To an adversary having previous knowledge about the embedding of algebraic predicates in the program, static analysis attack over the obfuscated code will simply be reduced to code pattern matching. *Opaquely non-deterministic predicates* are based on some function parameter selected at random [6]. The stealthiness of these predicates will be increased greatly if their random integers are drawn from a probability distribution which resembles the values of integer constants typically observed in programs.

Collberg et al. [1] used the intractability property of pointer aliasing to construct *aliased opaque predicates.* Their construction is based on the fact that it is impossible for approximate and imprecise static analysers to detect all aliases all of the time [7]. The basic idea is to construct a dynamic data structure and maintain a set of pointers on this structure. Opaque predicates can then be designed using these pointers and their outcome can be statically determined only if precise inter-procedural alias analysis can be performed on this complicated data structure. Collberg et al.'s definition of the resilience of an opaque predicate does not take into account dynamic analysis attacks. If an attacker can monitor the heap, registers, etc. during execution, then it may be revealed that a given predicate always evaluates to true or false.

Palsberg et al. [8] observed that in order to protect against a dynamic debugging attack, the obfuscator needs to avoid a predicate from evaluating to the same result. They proposed using *dynamically opaque predicates*, which are a family of correlated predicates which all evaluate to the same result in any given run, but in different runs they may evaluate to different results. It is an open problem to construct correlated dynamic opaque predicates and will be part of our future research. These predicates could have the following structure:

$$\texttt{if } (\Phi_1)\ S_1\ ;$$
$$\texttt{if } (\neg\Phi_2)\ S_1'\ ;$$

where S_1 and S_1' are variant versions of the same code block which are difficult to merge. A possible attack for this transformation is to prove that $S_1 \equiv S_1'$ and $\Phi_1 \Leftrightarrow \Phi_2$. If this is the case, then this obfuscated structure may be replaced by S_1 under certain conditions (such as S_1 does not change the value of $\neg\Phi_2$).

In [9], Majumdar and Thomborson suggested creating *temporally unstable opaque predicates* in a distributed environment of concurrently executing mobile agents from respective copies of aliased data structure values. A temporally unstable opaque predicate can be evaluated at multiple times at different program points during a single program execution such that the values observed to be taken by this predicate are not identical. There are a couple of advantages of making opaque predicates temporally unstable. The first one concerns its reusability; one predicate can be reused multiple times to obfuscate different control flows. Secondly, they are also resilient against static analysis attacks since their values depend on dynamically changing message communication patterns between participating agents. In addition to their resilience against static analysis attacks, temporally unstable opaque predicates are also difficult to attack by dynamic monitoring. For mounting a dynamic attack, the authors assumed an adversary capable of monitoring local states of individual agents through some malicious sniffer agent. Even in this adversarial model, it was argued, using the results of Chase and Garg [10], that constructing a global state from these partially observed agent local states is a computationally intractable problem.

4 Discussion and Future Work

In this brief survey, we highlighted the dispatcher model and opaque predicates for control-flow obfuscation. We also noted that obfuscatory strength cannot be guaranteed, in part because it is not known how to arbitrarily generate hard problem instances. Furthermore, the techniques which use hard complexity results as their theoretical basis are "wrong-way" reductions, from a complexity-theoretic perspective. These reductions explain why we should not expect to have exact static analysis tools that will work on all programs, however they do not prove the hardness of deobfuscating any specific output of any specific obfuscation system. Even so, an obfuscation system will have great practical importance if it resists all known attacks for at least as long as it would take to replace an obfuscated program by a differently-obfuscated program.

References

1. Collberg, C., Thomborson, C., and Low, D.: A Taxonomy of Obfuscating Transformations. Technical Report#148. 36 pp. Department of Computer Science, The University of Auckland, New Zealand. 1997.
2. Collberg, C., Thomborson, C., and Low, D.: Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In Proceedings of 1998 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98). Pages 184-196. 1998.
3. Drape, S.: Obfuscation of Abstract Data Types. DPhil thesis. Computing Laboratory. Oxford University. England. 2004.
4. Wang, C., Hill, J., Knight, J.C., and Davidson, J.W.: Protection of software-based survivability mechanisms. In Proceedings of the 2001 conference on Dependable Systems and Networks. IEEE Computer Society. Pages 193-202. 2001.
5. Chow, S., Gu, Y., Johnson, H., and Zakharov, V.A.: An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. In the proceedings of 4^{th} International Conference on Information Security, LNCS Volume 2200. Pages 144-155. Springer-Verlag. Malaga, Spain. 2001.
6. Venkatraj, A.: Program Obfuscation. MS Thesis. Department of Computer Science, University of Arizona. 2003.
7. Horwitz, S.: Precise Flow-insensitive may-alias in NP-hard. In ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 19 No. 1. Pages 1-6. 1997.
8. Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao, Q., and Zhang, Y.: Experience with software watermarking. In Proceedings of 16^{th} IEEE Annual Computer Security Applications Conference (ACSAC'00). IEEE Press. p308. New Orleans, LA, USA. 2000.
9. Majumdar, A. and Thomborson, C.: Manufacturing Opaque Predicates in Distributed Systems for Code Obfuscation. In Proceedings of the 29^{th} Australasian Computer Science Conference (ACSC'06). Pages 187-196. ACM Digital Library. Hobart, Australia. 2006.
10. Chase, C. and Garg, V.K.: Detection of global predicates: Techniques and their limitations. In the Journal of Distributed Computing, Volume 11, Issue 4. Pages 191-201. Springer-Verlag. 1995.