Editor: Mohamed Kaâniche, mohamed.kaaniche@laas.fr

Assessing the Security Properties of Software Obfuscation

Hui Xu and Michael R. Lyu | The Chinese University of Hong Kong

Software obfuscation traditionally refers to two kinds of problems: *general software obfuscation* and *cryptographic obfuscation*.[1]

General software obfuscation aims to make a software executable as unintelligible as possible, such that adversaries would have trouble understanding the program logic. It can be performed with lexical transformation, control transformation, data transformation, and so on. An example of such a software obfuscation tool is Obfuscator-LLVM.

Cryptographic obfuscation specifically aims to hide the secret keys embedded in software. For example, a point function ($\boldsymbol{I}_p(x) = 1$, if $x = p$, or 0 otherwise) can be cryptographically obfuscated by transforming p (the key) with a hash function, such that attackers can't determine the input that would lead $\boldsymbol{I}_p(x)$ to output 1. If cryptographic obfuscation is strong enough, it can serve as a basis for fancier cryptographic applications, such as homomorphic encryption.[2]

Both obfuscation approaches are important in practice. For example, licensing is a mechanism that controls the right to use software features. Usually, a restricted or trial software version is developed by starting with the full version and then controlling certain features by licensing, which validates certain keys for using the features. To prevent attackers from bypassing these controls, obfuscation can hide the license verification code; the point function is useful for hiding the keys embedded in the software. However, pure cryptographic obfuscation isn't enough because attackers might bypass the checking by modifying the executables (for example, by disabling the key checking code by jumping directly to the restricted feature code); hence, software obfuscation transformations such as control-flow obfuscation, which complicate the code itself to deter such modifications, are needed.

Obfuscation with Valid Security Properties

According to Boaz Barak and his colleagues' definition, an obfuscator $\boldsymbol{O}$ can be defined as a "compiler" that inputs a program $\boldsymbol{P}$ (represented as a circuit or a Turing machine) and outputs a new program $\boldsymbol{O}(\boldsymbol{P})$. The obfuscated program $\boldsymbol{O}(\boldsymbol{P})$ should possess the same functionality as $\boldsymbol{P}$, have the same efficiency as $\boldsymbol{P}$, and hold some unintelligibility properties.[2] Note that we use efficiency to denote a polynomial relationship in program size or computation time.

Virtual Black-Box Property

The ideal property of unintelligibility is the virtual black-box property, which means that $\boldsymbol{O}(\boldsymbol{P})$ leaks no information about the original program, or that attackers can't take advantage of $\boldsymbol{O}(\boldsymbol{P})$ other than as oracle access to the program. One exemplary program that can be black-box obfuscated is the point function. However, Barak and his colleagues showed that at least one family of programs (distinguisher programs that can discriminate some one-way functions) can't be black-box obfuscated.[2] This implies that we can't construct a universal obfuscator for all programs.

Indistinguishable Property

The black-box property isn't universally attainable, yet we still need valid security properties. A weaker notion is indistinguishable property: if two programs $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ are equivalent in both functionality and size, then $\boldsymbol{O}(\boldsymbol{P_1})$ and $\boldsymbol{O}(\boldsymbol{P_2})$ should leak exactly the same amount of information to attackers. Building indistinguishable obfuscated programs isn't difficult; for example, if a program class $\boldsymbol{P}$ has an efficiently computable canonical form (a uniform representation for all programs in the class), the computation of that canonical form already meets the indistinguishable property. Nevertheless, determining whether there are efficient indistinguishable obfuscators for all programs—and, if so, how to construct them—is challenging. Because the indistinguishable property provides no a priori guarantee of information hiding, another open question to explore is its security effectiveness when applied in different obfuscation scenarios.

Best-Possible Property

A security property that can enhance the indistinguishable property is the best-possible property. This requires that for any efficient learner $\boldsymbol{L}$, there exists an efficient simulator $\boldsymbol{S}$, such that $\boldsymbol{L}(\boldsymbol{O}(\boldsymbol{P_1}))$ and $\boldsymbol{S}(\boldsymbol{P_2})$ are functionally equivalent. Shafi Goldwasser and Guy Rothblum showed that the best-possible property is equivalent to the indistinguishable property but excludes obfuscators that can't efficiently obfuscate programs.[3]

Tools for Obfuscation with Valid Security Properties

Now we discuss potential techniques for building resilient obfuscators that can achieve valid security properties. Generally, an obfuscating approach with a valid security property should involve hard problems that attackers must solve, such that the difficulty of the problem can be used to measure the attacking complexity, or the obfuscation's security strength. There are two ways of doing this: mathematical approaches (such as multilinear jigsaw puzzles), which are generally considered for cryptographic obfuscators, and software analysis approaches (such as alias analysis), which are applicable for general software obfuscators.

An obfuscating approach with a valid security property should involve hard problems that attackers must solve.

Mathematical

Multilinear jigsaw puzzles are an application of multilinear maps on bounded-width branching programs. Sanjam Garg and his colleagues showed that such puzzles might be a candidate tool for constructing an indistinguishable obfuscator for all programs. This is because the hardness assumption states that the two output distributions of the jigsaw puzzle generator should be computationally indistinguishable.[4]

Software Analysis

Alias analysis attempts to statically determine whether two pointer expressions refer to the same memory location. Compiler optimizers can apply it to perform constant propagation and dead-code elimination. However, because aliasing can occur at any point during program execution, aliasing analysis is an undecidable problem in nature. For example, aliasing can occur conditionally (that is, *may-alias*): two pointer expressions might or might not refer to the same storage location depending on certain conditions. Moreover, the memory space's granularity might affect the precision of alias analysis, as memory space can be dynamically relocated. Therefore, alias analysis requires flow-sensitive analysis that computes what the memory location's pointer expressions refer to during each program point. Flow-sensitive analysis is expensive in terms of computation time.

Problems such as performing interprocedural may-alias analysis on multiple level pointers can be as hard as problems solved in polynomial time using a nondeterministic Turing machine (that is, NP problems). This means that although whether or not two pointers refer to the same location can be verified efficiently, it can't be calculated efficiently. However, the hardness can be compromised easily if not utilized properly. For example, Toshio Ogiso and his colleagues proposed an obfuscation approach based on such hardness incurred by pointer analysis.[5] But their obfuscated code example could be easily attacked by simplifying the clumsy point to possibilities with symbolic execution.

Deobfuscation Difficulty

Deobfuscation is the reverse of obfuscation—it transforms the obfuscated software to an explicit version that's easy to read.

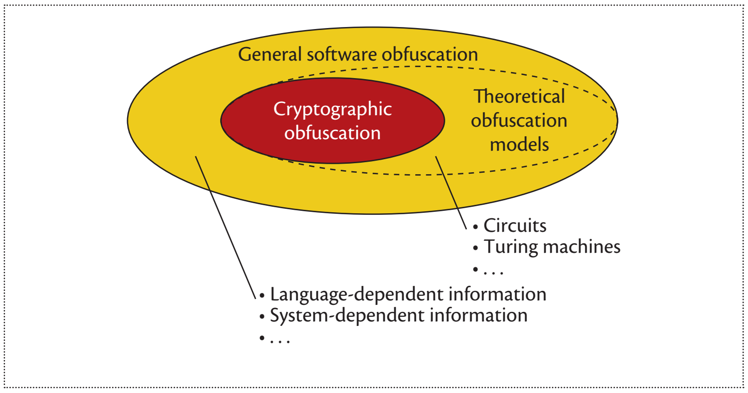
General software obfuscation
Cryptographic obfuscation
Theoretical obfuscation models
• Circuits
• Turing machines
• . . .
• Language-dependent information
• System-dependent information
• . . .

Figure 1. Relationships among different obfuscation concepts.

Deobfuscation's computational complexity is thus another important factor that can be used to measure obfuscation's security.

The deobfuscation process shouldn't change the obfuscated software's functionality. Hence, a precise deobfuscation transformation requires semantic equivalence verification. However, checking whether two programs are equivalent is a hard problem because, according to Rice's theorem, it's generally impossible to use static analysis to decide whether a program exactly computes a function.[6] Further research has concluded that deobfuscation is as hard as NP under specific models. However, such deobfuscation models require recovering the obfuscated program to its original version as much as possible, whereas a real attacker might not need to reverse $O(P)$ to its original version but rather to any version of P that leaks the secret information. For this reason, the deobfuscation problems discussed in the literature are much more difficult than most reverse-engineering tasks. Novel deobfuscation models with practical meanings are needed.

Gaps to Be Bridged

Why is achieving valid security properties for practical obfuscators so difficult?

First, the obfuscation concepts discussed in the literature aren't equivalent. As Figure 1 shows, cryptographic obfuscation is a subset of general software obfuscation; the theoretical obfuscation models with circuits or Turing machines can model cryptographic obfuscation problems well but not general software obfuscation problems. Circuits or Turing machines address simple mathematical gates such as ADD, SUB, AND, and OR but don't consider other high-level programming language or system-dependent information, such as the standard function calls in libc. Although some high-level function calls can be replaced by low-level instructions, this incurs much overhead and isn't recommended in modern program paradigms. Moreover, such high-level function calls are essential targets for reverse-engineering practical programs, which might make the security property of the obfuscation algorithm useless.

Second, although cryptographic obfuscation has achieved positive results (such as point function), it doesn't generally apply to software obfuscation because these two domains might define a successful attack differently. Taking the licensing mechanism as an example, a successful cracking implies key leakage from the view of cryptographic obfuscation, while practical adversaries might only need to locate the code that bypasses the license verification. In other words, cryptographic obfuscation assumes less powerful adversaries than general software obfuscation.

Finally, the best attainable security property—the indistinguishable property—is too weak to meet practical obfuscation requirements. An extreme case is that even if the secret isn't well-hidden in the obfuscated software, it might still qualify as an indistinguishable obfuscation property. Moreover, determining how to compose a practical obfuscator with this weak security guarantee is difficult.

Mitigating these gaps for practical software obfuscators is challenging. We propose rethinking the meaning of a successful attack on obfuscated software. We suggest possibly attainable security properties that are meaningful for practical software obfuscation scenarios, such as some properties against specific deobfuscation techniques, rather than general and weak properties, such as the indistinguishable property. In this way, considering or even obfuscating the language- and system-dependent information under such new adversary models would be much easier when feasible. ■

Acknowledgments

This work was supported by the National Natural Science Foundation of China (project 61332010), Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK 14205214 of the General Research Fund), and 2015 Microsoft Research Asia Collaborative Research Program (project FY16-RES-THEME-005).

References

1. N. Kuzurin et al., "On the Concept of Software Obfuscation in Computer Security," *Proc. 10th Int'l Conf. Information Security* (ISC 07), 2007, pp. 281–298.

2. B. Barak et al., "On the (Im)possibility of Obfuscating Programs," *J. ACM*, vol. 59, no. 2, 2012, article 6.
3. S. Goldwasser and G.N. Rothblum, "On Best-Possible Obfuscation," *Proc. 4th Theory of Cryptography Conf.* (TCC 07), 2007, LNCS 4392, Springer, pp. 194–213.
4. S. Garg et al., "Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits," *Proc. 54th IEEE Ann. Symp. Foundations of Computer Science* (FOCS 13), 2013, pp. 40–49.
5. T. Ogiso et al., "Software Obfuscation on a Theoretical Basis and Its Implementation," *IEICE Trans. Fundamentals of Electronics Communications and Computer Sciences*, vol. E86, no. A(1), 2003, pp. 176–186.
6. J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed., Addison-Wesley, 2006, pp. 397–399.

Hui Xu is a PhD student in the Computer Science and Engineering Department of The Chinese University of Hong Kong. Contact him at hxu@cse.cuhk.edu.hk.

Michael R. Lyu is a professor in the Computer Science and Engineering Department of The Chinese University of Hong Kong. Contact him at lyu@cse.cuhk.edu.hk.