On Program Security and Obfuscation

Josh MacDonald

December 10, 1998

1 Problem Statement

This project was set out to survey work in program protection and related fields. I investigate previous work in an attempt to unify a problem statement, along with the necessary and sufficient conditions for various forms of software protection. Past research contains a number of informal protection schemes and only a few formal attempts. The fact that I am unable to find an appropriate formal basis for this type of research has made progress difficult, so I have produced nothing more than this survey and a few speculations.

The general problem of program protection is widely held to be impossible. There is no doubt that the problem is difficult, but a formal result, whether positive or negative, is necessary. Given these difficulties, I hope only to convince the reader of the inadequacy of the current state of research for attempting to prove a result of this kind, and that program protection is an important, wide-open problem.

The problem, as I gather it, crosses cryptography with the fundamentals of programming languages. There exists a wealth of mathematical tools for secure communication, authentication, computation, and much more in the cryptography literature. The past 30 years of research in this field has produced amazing results–and these techniques have been rapidly integrated into computer systems. Program protection involves a similar set of goals, and many of the same techniques may apply once program protection is properly understood.

I will begin with by analyzing the various threats that people wish to protect against (§2). I will then discuss a recent, promising direction towards a solution (§3), summarize relevant topics in cryptography (§4) and relevant topics in program obfuscation (§5), discuss adversarial weaknesses in today's program analysis technology (§6), and summarize related work (§7).

2 Threat Model

Cryptographic techniques have been studied for insuring message encryption, authentication, integrity, and nonrepudiation. Each of these interests carries over to the protection of programs. For the purposes of this paper, I will restrict my terminology to programs which may be executed non-interactively, and without requiring special purpose or tamper-proof hardware. As usual, the participants Alice and Bob will be used to describe the distribution of a program in some form. Alice has written a program P and would like to publish it so that Bob may execute it on his operating system. Securely distributing a program is covered by existing techniques, so we may assume that Bob has securely received a message from Alice containing the program in its executable form. Bob must then be able to execute the program without requiring interaction on Alice's behalf—secure, interactive program execution is a much easier problem.

There are several types of security that each party is interested in, from which a definition of *perfect program security* can be obtained (even if it is unattainable). I will also discuss attacks by a *program cryptanalyst*, who would like to compromise a program's security for various purposes. All participants are polynomially bounded.

A very simple program model will be used. A valid program is expressed in some simple machine language, accepts inputs in domain D, and terminates with a value in some range R.

Few trust assumptions are acceptable. The operating system may not always trust the program–this has been studied. Hopefully, Bob trusts his operating system. Alice and her program, however, may not. Both Alice and the operating system may not trust Bob (or users presumed to be Bob), and it may be possible for Alice to help the operating system prevent attacks from a user with a detailed knowledge of the program, such as the classic buffer-overrun style of runtime code insertion attack.

By giving Bob a program, Alice has granted him the capability of running the program on a polynomial number of inputs. He knows, consequently, exactly how to compute an output for every input he can produce. This appears to be the greatest limitation in program security, as Bob must be told how to compute on every input in the domain. This limitation, however, is not as great as it may at first appear to be. If one-way functions exist, for example, they have a similar property: knowledge of the computation being performed does not provide any information about the inverse of the function. The question is then posed: *what else may be concealed in or about a computation?*

There are several obvious mechanisms by which program security might be achieved. First, there is control over the exact computation being expressed. The program needs only to be observably equivalent to its original specification, so a translator has many freedoms in arranging and introducing code. Additional, arbitrary computation may added, along with new inputs. These inputs might, for example, be used to challenge a program, preventing against replay attacks. In addition, an increase in the time and space the program requires to execute is acceptable in exchange for security. Second, a *program signature* may be attached when necessary for interpreting and verifying the inputs and outputs of a program.

There are a variety of attacks on unprotected programs, some of these are:

- A buffer overrun can allow a malicious user to introduce arbitrary code at runtime.

- Demand paging over an untrusted network, or execution from untrusted memory, allow an attacker to modify the executable at runtime.
- Redistribution of tampered or infected programs allows for software piracy and the introduction of computer viruses.
- Even redistribution of untampered code is problematic, since there are few techniques for proving authorship or ownership of a program.
- A mobile agent may desire to execute on untrusted hardware, and is open to many attacks from a malicious host.
- Reverse engineering of unprotected programs perhaps allows competitors to learn trade secrets.

Some of these are more difficult to protect against, but not all of these goals do seem impossible. Protecting against reverse engineering seems the most difficult task since the program contains all the necessary information to reconstruct its algorithms, but the others attacks may be preventable since they only depend on the *detection* of tampering. I will first sketch a solution, then detail how such a solution, if attainable, is sufficient to prevent against the some of the above attacks.

3 The Undetachable Program Signature

A program signature is said to be "undetachable" if it is possible to produce a program which signs its output in such a way that the signing code cannot be detached from the actual code. I attribute this concept to Sander and Tschudin [18], though the idea is roughly present in work by Collberg and Thomborson on software watermarking [8]. If a program signature could allow for the integrity of a computation to be verified, then tampering can be detected.

Suppose a pair of one-way functions can be generated with a trapdoor: s and s^{-1} such that neither is invertible. For a traditional digital signature, s is the public key and s^{-1} the private key—the signature is generated with the private key and verified with its inverse, the public key. Suppose that a program p can be composed with s^{-1} in such a way that neither can be recovered from the resulting program. That is, the resulting program is $p' = p \cdot s^{-1}$, and its results may be verified with the program's public key, s. In other words, Bob can learn how to compute the signature of the program's results, but not the signature function itself, so he cannot modify the behavior of f without knowing the inverse of s, which is presumed to be one-way. Of course, this is very speculative.

3.1 Replay Attacks

Knowledge of the legitimate signature for a certain program result should not allow the result to be replayed. A trusted operating system can be used to prevent against replay attacks by adding a challenge-response protocol to program execution. By adding an additional *execution key* program input to the program, inserting additional code to compute a one-way function on the execution key, and of course signing the result, the one-way function can be published along with the program's public key to verify the integrity of the challenge.

This mechanism greatly reduces the chances of a successful buffer-overrun or page-replacement attack, since the attacker must know how to produce not only the signature, but also the result of the one-way function which may have been in an arbitrary state of computation when the attack occured.

Since a mobile agent cannot trust its operating system to prevent replays, another mechanism must be used to prevent replays.

3.2 One-Way Functions on Case Statements

One potential mechanism for aiding prevention against reverse-engineering is to statically compute a one-way function on the value of constant case expressions. A boolean expression of the form $x = c$ for variable x and constant c can be replaced by $f(x) = f(c)$ where f is a one-way function. To completely reverse-engineer such a program would require a finding a set of code-covering inputs, which is known to be hard in the worst case. This idea is due to Manuel Blum.

3.3 Software Watermarking

The goal of software watermarking is to mark software with a secret message that is undetectable to the casual observer, may be reliably located, is difficult to remove by sound program transformations, and does not greatly affect program performance. Such technology can be used to deter the theft of program code, since a copyright message can be embedded in the watermark and used to prove theft. The first intellectual treatment of software watermarking appears in [8], and uses the structure of heap-allocated data to encode the watermark. In many ways, the goal of software watermarking is opposite the other goals presented here, since it attempts to conceal a message which may be easily extracted, even when tampering has occured. Still, watermarking is an important direction for intellectual property protection.

4 Cryptographic Tools and Approaches

Sander and Tschuden briefly examine the security of an undetachable signature scheme which relies on the difficulty of decomposing two rational functions [18]. The problem has already been studied and has no known polynomial time algorithm. They then give several attacks, an improvement, and leave the problem open for future work.

Several cryptosystems use concepts similar to these. For example, the McEliece public-key encryption algorithm composes a random permutation matrix with a certain type of error correcting matrix [19]. Encryption relies on knowing the product of the matrices, while decryption relies on knowing the inverse of the permutation matrix and the original error correcting matrix. A more recent, but similar, public key cryptosystem uses the difficulty of finding the closest lattice point to some vector [13], a problem which is proven to be hard in the average case [3]. The public key is a highly non-orthogonal lattice basis, while the private key is an equivalent, but reduced, basis. These cryptosystems are attractive because their computational model is more expressive than the standard number-theoretic approaches, and they more efficient as well.

More generally, there are results stating that any trapdoor function can be used for signing with provably high security [5]. This is in contrast to classical approaches which rely on the difficulty of factoring numbers or other number-theoretic problems (without provable difficulty). Fundamental results of this nature are useful in attempting to integrate the theory into other problem domains. One-time (or finite-use) signature schemes are another approach to signing that does not require a trapdoor function, only a one-way function, but these can only be used when the number of signatures is bounded. Still, this technique makes all one-way functions available for computing signatures. Since the security of the decomposition approach above may depend on the stealth of a particular computation, how indistinguishable it is from the real computation, this flexibility is attractive. For example, there are one-way functions based upon DAG and tree traversals [6] that have a very natural integration into pointer-rich programs.

5 Program Obfuscation Approaches

The goal of program obfuscation is to make a program difficult to understand. Open to interpretation, this goal refers to the human or machine difficulty of understanding various program properties and the algorithms contained within. There have been many ad hoc techniques proposed; a taxonomy of approaches is discussed in [9]. Since the hope is that program protection rests upon the ability to conceal something in a program which cannot be extracted, these techniques informally suggest a direction. We are in search of security, however, and not obscurity, so a formal proof of security is a necessity. Obfuscation transformations have been studied in terms of their cost and power, but there is no formal basis for making claims about the difficulty of understanding a program, since it is a very cognitive measure. Instead, claims can be made about the difficulty of algorithmically undoing a certain program obfuscation. The models which have been given, however, are not very satisfying for the purpose of making claims about security. We would like a guarantee, in terms of some security parameter, of the intractability of signing a result not produced by an untampered execution of the program.

Collberg et al. have analyzed the difficulty of undoing program obfuscating transformations by looking to the worst-case complexity of the program analyses required to provide information necessary to undo them [10]. For example, the difficulty of undoing a transformation which inserts false control paths depends on the difficulty of alias or shape analysis to statically determine a predicate's truth value [10]. This suggests the use of hard problems in program analysis to conceal program artifacts. A potential proof for some solution to the undetachable signature problem described above might show that it is necessary to compute some accurate program analysis and that, by construction, the difficulty of computing that analysis has an average-case difficulty which is exponential in some security parameter. Pointer (may-) alias analysis is an appealing problem because it is fundamental to many other program analyses which require this information for soundness and for computing control- and data-dependence.

It is already proven that many program properties are undecidable in the presence of loops and dynamic allocation. Pointer alias analysis is one of these [15] [17]. Even accurate, flow-insensitive alias analysis is NP-hard [14]. Thus, pointer alias analysis might be leveraged for the protection of programs if a problem can be stated in terms which necessitate accurate alias analysis. Broadly speaking, this suggests studying the composition and decomposition of recursive functions from a computability-theoretic point of view. Adleman gives an interesting, formal discussion of computer viruses along these lines [2], and proves that detecting viruses is undecidable.

6 The Program Analysis Adversary

The construction of programs which are difficult to analyze has not, to my knowledge, been formally studied. Many of the problems listed above required only program tamper-detection, but the remaining few rest upon a program's resilience to reverse engineering—on its ability to resist understanding. Even if this concept cannot be formalized, it is valuable. Consider the following, informal argument.

Once again, Alice sends a program to Bob. Alice must reveal a computation which produces the desired results. Alice should attempt to send a program with as little helpful information as possible. How much information must she reveal to Bob? It is likely that Bob is interested in modifying the computation in some goal-oriented way, to integrate it into his own product, for example, or to defeat a copy-protection scheme.

Bob is looking to construct a program similar to the one Alice supplied, so he would like to have a high-level representation of the program in which his modifications can be localalized. There are an uncountable number of program representations which perform the same function, though, and some of these will be less helpful to Bob than others. It should be possible for Alice to express her function, therefore, so that it looks "random" to Bob, where "random" needs a firm definition, but intuitively relates a program representation to the density of nearby representations which are similar in some high-level representation, as opposed to the number of unrelated computations nearby. A simple compiler transforms a high-level program into one which nearly resembles the high-level program, and similar programs will appear very close in the simple executable program representation [7]. An obfuscating transformation attempts to remove information from and alter the program so that the distribution of similar programs is not so dense, making Bob's task harder. Artificially increasing program size naturally decreases this density.

6.1 One-Way Program Transformations

Alice would like to give Bob as little information as possible. The undecidability of aliasing implies that some approximation must always be done [15]. If an analysis must be approximate, it stands to reason that an adversary can construct a program in control of the quality and difficulty of various approximations. A program transformation is one-way when it is not invertible, so they are appealing because they make recovery of the original program representation difficult. Some layout transformations are clearly one-way, such as the replacement of a program's identifiers by random symbols. Collberg et al. list various others, including the reordering of statements and expressions and function inlining [9], but these transformations are not very effective. Transformations based on eliminating approximate program analysis results might allow stronger statements to be made.

One simple case is a transformation intended to eliminate information that might be gained by a path-insensitive may-alias analysis. A path-insensitive program analysis considers all permutations of statement orderings to avoid considering an exponential number of program paths [12]. The alias analysis typically construct a *points-to graph*, which contains the possible aliasing relations amongst equivalence classes of program locations. By making nodes (program locations or sets of program locations) in the points-to graph strongly connected, these algorithms must conclude that each program location may-alias every other program location. This can be accomplished with the use of global variables and the insertion of dead statements, as in [10]. Of course, dead statements may be detected, but the general problem is undecidable [20]. However, the goal of eliminating aliasing information was to confound other analyses, so locating statically dead statements is all the more difficult without accurate alias information.

7 Related Work

Another solution for securing mobile code is to use trusted, secure hardware devices. Yee and Tygar discuss the use of secure coprocessors for a variety of applications in electric commerce and program protection [21]. A theory of secure computation for machines with some finite number of secure registers and otherwise insecure RAM has been developed by Goldreich and later improved by Ostrovsky [16]. These solutions have succeeded, and tamper-proof, secure hardware is available today, though the solution requires wide replacement of existing hardware and infrastructure. Another cryptographic solution to software protection requires only securing a small portion of the instructions being read to detect tampering [11].

Many types of interactive, secure, multiparty computation have been studied. Secret-sharing schemes distribute various forms of computation, trust, and arbitration amongst multiple machines. These protocols do not directly apply to the problem at hand since they require interactivity.

Secure protocols for evaluating boolean circuits have been studied in which one participant has a function, the other participant has a value, and neither participant wishes to share information with the other [1].

A tamper-resistant software scheme has been proposed by Aucsmith which relies on obfuscation via lazy self-decryption, self-modifying code, and sharing integrity checks between modules to make localized tampering difficult [4].

8 Conclusion

I have surveyed a number of issues related to tamper-detection for program execution. This is a an important problem for system security, and, though it is perceived to be very difficult, no formal theory has been developed to prove or disprove the existence of such a scheme. The problem domain crosses topics in cryptography with programming languages and their analysis. It remains an open problem.

Acknowledgements I would like to thank Manuel Blum and David Wagner for their assistance.

References

[1] Abadi, M., and Feigenbaum, J. Secure circuit evaluation. *Journal of Cryptology 2*, 1 (1990), 1–12.

[2] Adleman, L. M. An abstract theory of computer viruses. In *Advances in Cryptology — CRYPTO'88* (1988), S. Goldwasser, Ed., Lecture Notes in Computer Science (No. 403), Springer Verlag.

[3] Ajtai, M. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing* (Philadelphia, Pennsylvania, May 1996), pp. 99–108.

[4] Aucsmith, D. Tamper resistant software: An implementation. In *Information Hiding 1996* (1996), R. J. Anderson, Ed., no. 1174 in LNCS, SV, pp. 317–333.

[5] Bellare, M., and Micali, S. How to sign given any trapdoor permutation. *Journal of the Association for Computing Machinery 39*, 1 (Jan. 1992), 214–233.

[6] BLEICHENBACHER, D., AND MAURER, U. M. Directed acyclic graphs, one-way functions and digital signatures. In *Proc. CRYPTO 95* (1994), Y. G. Desmedt, Ed., Springer, pp. 75–82. Lecture Notes in Computer Science No. 839.

[7] CIFUENTES, C., AND GOUGH, K. J. Decompilation of binary programs. *Software—Practice and Experience 25*, 7 (July 1995), 811–829.

[8] COLLBERG, C., AND THOMBORSON, C. Software watermarking: Models and dynamic embeddings. In *Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (to appear)* (Jan. 1999).

[9] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. Tech. Rep. 148, Department of Computer Science, University of Auckland, 1996.

[10] COLLBERG, C., THOMBORSON, C., AND LOW, D. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, Jan. 1998), pp. 184–196.

[11] DOMINGO-FERRER, J. Software run-time protection: A cryptographic issue. In *Advances in Cryptology—EUROCRYPT 90* (21–24 May 1990), I. B. Damgård, Ed., vol. 473 of *Lecture Notes in Computer Science*, Springer-Verlag, 1991, pp. 474–480.

[12] FOSTER, J. S., FÄHNDRICH, M., AND AIKEN, A. Flow-insensitive points-to analysis with term and set constraints. Tech. Rep. UCB/CSD-97-964, EECS Department, University of California, Berkeley, Aug. 1997.

[13] GOLDREICH, GOLDWASSER, AND HALEVI. Public-key cryptosystems from lattice reduction problems. In *Electronic Colloquium on Computational Complexity, technical reports.* 1996.

[14] HORWITZ, S. Precise flow-insensitive may-alias analysis is NP-Hard. *ACM Transactions on Programming Languages and Systems 19*, 1 (Jan. 1997), 1–6.

[15] LANDI, W. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems 1*, 4 (Dec. 1992), 323–337.

[16] OSTROVSKY, R. An efficient software protection scheme. In *Advances in Cryptology: CRYPTO '89* (Berlin, Aug. 1990), Springer, pp. 610–611.

[17] RAMALINGAM, G. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems 16*, 5 (Sept. 1994), 1467–1471.

[18] SANDER, T., AND TSCHUDIN, C. F. Towards mobile cryptography. Tech. Rep. TR-97-049, International Computer Science Institute, Nov. 1997.

[19] SCHNEIER, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley and Sons, Inc., New York, NY, USA, 1994.

[20] WINSKEL, G. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing series. MIT Press, Feb. 1993.

[21] YEE, B., AND TYGAR, J. D. Secure coprocessors in electronic commerce applications. In *Proceedings of the first USENIX Workshop of Electronic Commerce: July 11–12, 1995, New York, New York, USA* (Berkeley, CA, USA, July 1995), USENIX Association, Ed., USENIX, pp. 155–170.