Revisiting Software Protection[⋆]

Paul C. van Oorschot

Digital Security Group, School of Computer Science
Carleton University, Canada
paulv@scs.carleton.ca

Abstract. We provide a selective survey on software protection, including approaches to software tamper resistance, obfuscation, software diversity, and white-box cryptography. We review the early literature in the area plus recent activities related to trusted platforms, and discuss challenges and future directions.

1 Introduction and Overview

Software protection has recently attracted tremendous commercial interest, from major software vendors to content providers including the movie and music recording industries. Their digital content is either at tremendous risk of arriving free on the desktops of millions of former paying customers, or on the verge of driving even greater profits through new markets. The outcome may depend in large part on technical innovations in software protection, and related mechanisms for digital rights management (DRM) - controlling digital information once it resides on a platform beyond the control of the originator. Privacy advocates are interested in similar mechanisms for protecting personal information given to others in digital format. Related activities include Microsoft's heavy investment in a next generation trusted hardware platform (Palladium) [41], and the recent award by the U.S. Air Force Research Laboratory of a US$1.8m (£1.13m) research contract involving software obfuscation [25].

Software protection falls between the gaps of security, cryptography and engineering, among other disciplines. Despite its name, software protection involves many assumptions related to hardware and other environmental aspects. A significant gulf currently exists between theory and practice. Inconsistencies have arisen in the relatively sparse (but growing) open literature as a result of differences in objectives, definitions and viewpoints. All of these issues provide research opportunities.

We provide a selective survey on software protection and related areas, and encourage further research. We offer a number of viewpoints, discuss challenges, and suggest future directions.

[⋆] Version: 15 July 2003.

1.1 Focus

Under the heading of software protection, our main focus is on techniques useful for protecting software from reverse engineering (by obfuscation), modification (by software tamper resistance), program-based attacks (by software diversity), and BORE – break-once run everywhere – attacks (by architectural design).

Protecting content typically requires protecting the software which processes the content – motivating our focus on software protection. We are interested in copy protection – a term many use interchangeably with software protection – to the extent that it requires many of the same protections against reverse engineering and software tampering. We do not pursue digital watermarks, a form of steganography (see Petitcolas et al. [44] for a taxonomy); they typically do not by themselves prevent attacks *a priori*, but may be used for tracking and finding violators after-the-fact, often in conjunction with legal remedies.

1.2 Organization

The remainder of this paper is organized as follows. §2 reviews some early literature on software protection. §3 discusses a selection of software protection approaches including software tamper resistance, software obfuscation, software diversity, and white-box cryptography. §4 provides observations related to positive and negative results on software obfuscation, the challenges of using complexity-theory as a basis for security analysis, definitions of efficiency, and security through obscurity. §5 reviews recent directions in the related area of enhancing platform security with low-cost "trusted" hardware and secure software boot-strapping. §6 provides concluding remarks.

2 Early Research on Software Protection

As usual, we find it instructive to review the earliest published works in any research area, to learn what was known to the original experts in the area, and has since been forgotten or re-invented.

One of the earliest published works in software protection is the 1980 thesis of Kent [37], which addresses the security requirements of software vendors: protection from software copying and modification (e.g. the latter by physical attacks by users, or program-based attacks). Tools proposed to address these requirements include physical tamper-resistant modules (TRMs) and cryptographic techniques; one approach involves using encrypted programs, with instructions decrypted immediately prior to execution. Kent also noted the dual problem of user requirements that externally-supplied software be confined in its access to local resources (cf. hostile host vs. hostile code, §3.2 below).

Gosler's software protection survey [31] examines circa-1985 protection technologies including: hardware security devices (e.g. dongles), floppy disc signatures (magnetic and physical), analysis denial methods (e.g. anti-debug techniques, checksums, encrypted code) and slowing down interactive dynamic analysis. The focus is on software copy prevention, but Gosler notes that the strength of resisting copying should be balanced by that of analyzing the software (e.g. reverse engineering to learn where to modify software, and for protecting proprietary algorithms) and that of software modification (to bypass security checks). Useful tampering is usually preceded by reverse engineering.

Gosler also notes that one should expect that an adversary can carry out dynamic analysis on the target software without detection (e.g. using in-circuit emulators and simulators), and that in such a case, as a result of repeated experiments, one should expect the adversary to win. The practical defense objective is thus to make such experiments "extremely arduous". Another suggestion [31, p.154] is cycling software (e.g. via some forced obsolescence) at a rate faster than an adversary can break it; this anticipates the paradigm of forced software renewal (cf. Jakobsson and Reiter [36], who propose discouraging pirates through forced updates and software aging). This approach is appropriate where protection from attacks for a limited time period suffices.

Herzberg and Pinter [33] consider the problem of software copy protection, and propose a solution requiring CPU encryption support (which was far less feasible when proposed almost 20 years ago, circa 1984-85). Cohen's 1993 paper [19] (see also §3.4 below) on software diversity and obfuscation is directly related to software protection, and provides a wealth of techniques and insights.

Goldreich and Ostrovsky's 1996 journal paper [30] (and earlier related individual 1987 and 1990 papers) provides one of the earliest theoretical foundation pieces. They reduce the problem of software protection – which they take to mean unauthorized software duplication – to that of efficient (in the theoretical sense) simulation on *oblivious RAMs*. A new issue they address is the extraction of useful information gained by an adversary examining the memory access patterns of executing programs. To address this, oblivous RAMs replace instruction fetches in the original program by sequences of fetches, effectively randomizing memory access patterns to eliminate information leakage. Interestingly, the subsequent practical tamper resistance system of Aucsmith [6,7] (see §3.3 below) addresses similar issues by a combination of just-in-time instruction decryption, and re-arranging instruction blocks at run-time to dynamically change the addresses of executing statements during program execution.

3 Software Protection Approaches

In this section we outline a selection of software protection approaches: obfuscation through code transformations (for protection against reverse engineering); white-box cryptography (for protecting secret keys in untrusted host environments); software tamper resistance (for protection against program integrity threats); and software diversity (for protection against automated attack scripts and widespread malicious software). We do not consider copy protection *per se*, but note that many of these techniques may serve useful roles in a copy protection solution. Approaches not discussed include the use of anti-debugging techniques.

3.1 Software Obfuscation via Automated Code Transformations

Several researchers have published papers on software obfuscation using automated tools and code transformations (e.g. Collberg et al. [21,22]). One idea is to use language-based tools to transform a program (most easily from source code) to a functionally equivalent program which presents greater reverse engineering barriers. If implemented in the form of a pre-compiler, the usual portability issues can be addressed by the back-end of standard compilers.

For design descriptions of such language-based tools, see Collberg et al. [20], Nickerson et al. [16], and C. Wang [52]. Cohen [19] suggested a similar approach already in the early 1990's, employing obfuscation among other mechanisms as a defense against computer viruses. Cohen's early paper, which is strongly recommended for anyone working in the area of software obfuscation and code transformations, contains an extensive discussion of suggested code transformations (see also Collberg et al. [20], and the substantial bibliography in the circa-2000 survey of Collberg et al. [23]).

C. Wang [52] provides an important security result substantiating this general approach. The idea involves incorporating program transformations to exploit the hardness of precise inter-procedural static analysis in the presence of aliased variables (cf. Collberg et al. [20, §8.1]), combined with transformations degenerating program flow control. Wang shows that static analysis of suitably transformed programs is NP-hard.

Collberg et al. [20] contains a wealth of additional information on software obfuscation, including notes on: a proposed classification of code transformations (e.g. control flow obfuscation, data obfuscation, layout obfuscation, preventive transformations); the use of *opaque predicates* for control flow transformations (expressions difficult for an attacker to deduce, but whose value is known at compilation or obfuscation time); initial ideas on metrics for code transformations; *program slicing tools* (for isolating program statements on which the value of a variable at a particular program point is potentially dependent); and the use of *(de)aggregation* of flow control or data (constructing bogus relationships by grouping unrelated program aspects, or disrupting legitimate relationships in the original program).

3.2 Untrusted Host Environment and White-Box Cryptography

The fairly well-studied, but still challenging, *malicious code problem* is as follows: how should a host machine be protected from potentially untrusted external code (e.g. downloaded from a web site, or arriving as an email attachment). Standard solutions include containment through sand-boxing, verification of source by code-signing, and anti-virus programs.

More closely related to software protection is the less-studied dual, the *malicious host problem*: how should a trusted program (e.g. containing a proprietary algorithm, private or critical data, or special access privileges) be protected from a potentially malicious host. This problem has received the most attention in the context of mobile code security (e.g. see Chess [14]; Sander and Tschudin

[46,45]; see also Algesheimer et al. [1]). Both problems were noted already more than 20 years ago by Kent [37] (see §2).

It is clear that the standard cryptographic paradigm – involving known algorithms, secret keys, and trusted communications end-points – fails entirely in the malicious host environment, as secret keys are directly visible. Moreover, as demonstrated by van Someren and Shamir [49], finding cryptographic keys in memory is quite easy – their randomness and size distinguishes them from other items in memory, which generally contain redundancy. Thus storing critical keys in memory is a vulnerability, given the ubiquity of malicious software.

The same malicious host issues arise in digital rights management applications where software attempts to constrain what end-users may do with content (e.g. music, movies and books), or with respect to modifying the software itself. Indeed, the host is effectively considered a hostile environment. Once a software vendor or content provider makes their digital product available on a system not under their control, virtually all such control over the digital item is indeed lost. The threats include manual attacks (humans including legitimate end-users, typically aided by software tools) benefiting from hands-on access to the executing software; and program-based attacks by malicious software, programs which exploit software vulnerabilities, or simple local or remote exploits of access control failures.

This leads to an extremely severe threat model: the *white-box attack context* of Chow et al. [17] (see also [18,35]) and *white-box cryptography* – cryptographic implementations designed to withstand attack in the white-box context. The white-box model contrasts traditional *black-box* models where only input-output or external behavior of software may be observed. An intermediate ground is *gray-box* attacks (also called *side-channel attacks* or *partial-access attacks*), such as fault analysis attacks (e.g. [10,11]) and the timing and power analysis attacks of Kocher and others, which involve the use of additional information.

The white-box attack context assumes fully-privileged attack software has full access to the implementation of cryptographic algorithms (e.g. shares the host), can view and alter internal algorithm details, and can observe dynamic execution involving instantiated cryptographic keys. The attacker's objective is to extract the cryptographic key, for use on a different platform. As suggested above, in this context standard cryptographic algorithms fail to provide protection. Here the choice (diversity) of implementation appears to be a remaining line of defense.

3.3 Software Tamper Resistance

Software obfuscation provides protection against reverse engineering, the goal of which is to understand a program. Reverse engineering is a typical first step prior to an attacker making software modifications which they find to their advantage. Detecting such integrity violations of original software is the purpose of software tamper resistance techniques. Software tamper resistance has been less studied in the open literature than software obfuscation, although the past few years has seen the emergence of a number of interesting proposals.

Fundamental contributions in this area were made by Aucsmith [6], in conjunction with Graunke [7] at Intel. Aucsmith defines *tamper resistant software* as software which is resistant to observation and modification, and can be relied upon to function properly in hostile environments. An architecture is provided based on an *Integrity Verification Kernel* (IVK) which verifies the integrity of critical code segments. The IVK architecture is self-decrypting and involves self-modifying code.

Working under similar design criteria (e.g. to detect single bit changes in software), Horne et al. [34] also discuss self-checking code for software tamper resistance. At run time, a large number of embedded code fragments called *testers* each test assigned small segments of code for integrity (using a linear hash function and an expected hash value); if integrity fails, an appropriate response is pursued. The use of a number of testers increases the attacker's difficulty of disabling testers.

In related work, Chang and Atallah [12] propose a scheme with somewhat broader capabilities involving a set of *guards* which can be programmed to carry out arbitrary tasks – one example is check-summing code segments for integrity verification providing software tamper resistance. Another suggested guard function is actually repairing code (e.g. if a damaged code segment is detected, downloading and installing a fresh copy of the code segment). They also outline a system for automatically placing guards within code.

Chen et al. [13] propose *oblivious hashing*, which involves compile-time code modifications resulting in the computation of a running *trace* of the execution history of a program. Here a trace is a cumulative hash of the values of a subset of expressions which occur within the normal program execution.

3.4 Software Diversity

Diversity is an idea which is part of the software folklore, but it appears to only recently have received significant attention in the security community. The fundamental idea is simple: in nature, genetic diversity provides protection against an entire species being wiped out by a single virus or disease. The same idea applies for software, with respect to resistance to the exploitation of software vulnerabilities and program-based attacks. In this regard, however, the trend towards homogeneity in software is worrisome: consider the very small number of different Internet browsers now in use; and the number of major operating systems in use, which is considerably smaller than 10 years ago.

The value of software diversity as a protection mechanism against computer viruses and other software attacks was well documented by Cohen [19] already in 1992-93. The architecture of automated code transformation tools to provide software obfuscation (see §3.1) can be modified slightly to provide software diversity: rather than creating one new instance of a program which is functionally equivalent to the original (and hard to reverse engineer), create several or many. Here the difficulty of reverse engineering or tampering with a single program instance is one security factor, but a more important factor is that an exploit crafted to succeed on one instance will not necessarily work against a second. Forrest et al. [28] pursue this idea in several directions, including simple randomizations of stack memory to de-rail high-profile buffer-overflow attacks.

The idea of relying on diversity for improving the reliability and survivability of networks has gained recent popularity, subsequent to incidents of global terrorism (e.g. see C. Wang [52] for contributions and references). The value of diversity for security and survivability was also recognized in the 1999 *Trust in Cyberspace* report [47], among others.

4 Other Observations on Software Protection

The relative scarcity in the open literature of theoretical papers on software protection and obfuscation suggests that the field remains in its early stages. Not surprisingly, there exist inconsistencies in definitions, models, and conclusions in the existing literature, and often practical papers are entirely lacking in the former two. Often, the objectives of attackers are not clearly (if at all) defined, making security analysis of proposed solutions a vague pursuit. A further challenge is that for techniques whose security supposedly is based on the difficulty of solving hard problems, it is often unclear if attackers must necessarily solve the underlying difficult problems to achieve their objectives.

4.1 Positive and Negative Theoretical Results

On the side showing promise for software obfuscation are NP-hardness results of C. Wang [52] (see also the earlier report [51]), the related results of Ogiso [43], and PSPACE-hardness results of Chow et al. [15]. These results suggest that one may expect code transformations to significantly (e.g. exponentially) increase the difficulty of reverse-engineering, and provide the beginnings of a foundation justifying the approach and security possible through code transformations.

In constrast are the impossibility results of Barak et al. [9], who prove that the following device does not exist: a software *virtual black box generator* which can protect *every* program's code from revealing more than the program's input-output behavior. While on the surface this result is quite negative for practitioners interested in software obfuscation, upon deeper inspection this is not so (despite the rather suggestive title of the paper); the results simply arise from the choice of definitions, model, and question posed.

In practice, the non-existence of such a virtual black box generator would appear to be of little concern. Of greater interest are several different questions, such as: to what proportion of programs of practical interest does this result apply; do obfuscators exist which are capable of obfuscating programs of practical interest; and can a more practical model be defined, allowing some level of non-critical information to leak from the execution of a program, provided it is not useful information to the attacker.

4.2 Security Justified by Complexity Class Results

The usual caveats also exist regarding the use of complexity-theoretic arguments as the basis for security. While NP-complete problems [29] are generally considered intractable, this intractability is based on the difficulty of the hardest problem instances. However, some NP-complete problems are easy in the average case, while random instances of others are still difficult. Thus for use in security, of greater interest than worst-case is average-case complexity analysis (e.g. see the overview by J. Wang [50]). In fact, we require not only average-case difficulty, but the probability of easy instances arising being very low.

Moreover, there are many cryptographic examples where difficult problems have been used to justify the security of proposals, which were later proven to be easily broken due to the fact that the particular problem instances built into actual instantiations turned out, for various reasons, to be far weaker than the hardest, or average, problem instances.

By way of comparison, it is interesting to recall the definition of the preimage resistance property for cryptographic one-way hash functions (e.g. [40, p.323]): for *essentially all* pre-specified outputs, it is computationally infeasible to find any input which hashes to that output. Thus a one-way hash function is considered good if it is difficult to invert for almost all outputs. Note that the definition does not simply require that there exist *some* hard instances.

4.3 Definition of Efficiency

A separate challenge, that is not unique to the literature on software protection, is the difference between what theoreticians and practitioners refer to as efficient. In standard complexity theory, slowing down running time by "only" a polynomial factor leaves a polynomial-time algorithm in polynomial time; and logarithmic slowdowns are considered quite good. In constrast in practice, a slowdown by a constant factor of as little as two (2), let alone two orders of magnitude (constant 100), is often far more than can be tolerated. Indeed in some cases, an overall slowdown of 10-20% is intolerable. Similar comments apply for space (although in practice, except in constrained environments, memory is now typically far less of a barrier - especially for personal computers).

A related comment offers a more graphical illustration for running time: for any fixed key length t, a t-bit key space can be exhaustively searched in constant time 2^t; therefore, the time to break a strong cipher with a 128-bit key, e.g. AES [24], is uninteresting from a complexity-theoretic viewpoint – namely, $O(1)$.

4.4 Security through Obscurity Vs. Software Obfuscation

A frequently cited set of cryptographic principles devised by Auguste Kerckhoffs in 1883 [38] stipulates that security should not suffer if an adversary knows all the details of an encryption function aside from the secret key (and therefore, all security rests on the secrecy of this key). Amateur cryptographers often violate this rule, designing systems whose security relies on secrecy of design details, which invariably become known. Such "security through obscurity" is thus severely frowned upon. Due to the language similarity with the phrase "software obfuscation", many people are pre-conditioned to also frown upon software obfuscation, without thinking further. (Of course, depending on the type of software obfuscation, this may be entirely justified.)

In our view, software obfuscation can and should be fundamentally distinct from security through obscurity. To this end, software obfuscation techniques should be pursued in which there are a very large space of possible transformations of an original software program to a transformed program, and indeed the security should not be compromised by an adversary knowing the set of possible mappings (or having the software transformation tool itself available). The choice among possible mappings should be sufficiently large that the security rests on the wide range of possibilities (analogous to a large key space).

5 Low-Cost Trusted Hardware Approaches

Concerns have continued to mount regarding trust in the Internet as a reliable platform for secure e-commerce, and as a foundation for content protection in digital rights management applications. Behind these concerns are inherent limitations of software-only security solutions. Such issues have resulted in efforts to develop a low-cost, commercial generic trusted hardware platform. The Trusted Computing Platform Alliance (TCPA), recently renamed the Trusted Computing Group [48], began in January 1999 under the efforts of Compaq, HP, IBM, Intel and Microsoft. Version 1.0 of the TCPA specifications were released early in 2001, defining the core security funcationality of a trusted subsystem for computing platforms, intended to "create a foundation of trust for software processes, based on a small amount of hardware within such platforms" [8, p.5].

Microsoft has also launched a separate (related) initiative originally known as *Palladium*, and recently renamed the Next-Generation Secure Computing Base ([41,42]; see also patents by England et al. [26,27]). Lively debate is ongoing (e.g. see Anderson's FAQ [2]) about the good and evil which will arise as a result of either or both TCPA and Palladium, and whether or not erosion of end-user control of their own devices, due to corporate-controlled DRM, is a likely result.

Earlier, Arbaugh et al. proposed a general architecture of how to develop such a trust foundation ([4]; and related patent [5]), intializing a computer system by successively validating the integrity of each layer throughout the bootstrap process, thereby providing a chain of integrity guarantees based on initial trust in the hardware layer.

Lie et al. [39] explore an alternate solution to the failures to date of software-only tamper-resistance, examining the hardware implementation of *execute-only memory*. Such memory allows the execution of stored-memory instructions, but no other manipulations. Gutmann [32] provides a clear discussion of the security issues facing cryptographic implementations in software under general-purpose operating systems, and examines the design challenges in overcoming these issues by employing secure cryptographic co-processors. Anderson and Kuhn [3] provide a number of attacks against tamper resistant (especially low-cost) devices, and make the case that building and using such devices properly is deceptively difficult.

6 Concluding Remarks

The theoretical results to date on software obfuscation leave room for software protection of considerable practical value. This should be of no surprise – indeed, the impossibility of building a program to determine whether other software is malicious does not preclude highly valuable computer virus detection technologies, and a viable (even lucrative) anti-virus industry.

We believe that it is still early in the history of research in the areas of software protection and obfuscation, and that many discoveries and innovations lie ahead – perhaps especially in the areas of software diversity (which seems to be very little utilized presently), and software tamper resistance.

We expect to see more open discussion of specific techniques, and believe that, similar to the history in the field of cryptography, the surest way to obtain an increased number of secure techniques is to involve public scrutiny, peer evaluation, and open discussion in the literature. We see the past trends of proprietary, undislosed methods of software obfuscation techniques analogous to the early days in cryptography, where invention and use of (weak) unpublished encryption algorithms by novices was commonplace.

A factor in favour of those promoting software obfuscation, software tamper resistance, and related software protection methods is Moore's law. As computing cycles become yet faster and faster, and the availability and speed of memory continue to increase, the computational penalties typically experienced in relation to many software protection approaches, will be unimportant. (Again, this seems analogous to the execution of 512-bit RSA being intolerably slow on a PC 20 years ago.)

As has been the case for some time, one of the greatest challenges in this area remains the lack of analysis techniques, and metrics for evaluating and comparing the strength of various software protection techniques. As a first step before obtaining such metrics, we believe more work is necessary in clarifying the exact goals of software obfuscation, and the precise objectives of attackers. We also believe there is a substantial research opportunity to fill in the large gap between the practical and theoretical progress in this area. For techniques of practical interest, we see opportunities to define models and approaches better reflecting applications for which software protection of short durations suffices (cf. forced software updates, §2).

Acknowledgements: The author acknowledges the generous support of the National Sciences and Engineering Research Council of Canada for support under both a Canada Research Chair and an NSERC Discovery Grant.

References

1. J. Algesheimer, C. Cachin, J. Camenisch, G. Karjoth, "Cryptographic Security for Mobile Code", pp. 2–11 in Proc. 2001 IEEE Symposium on Security and Privacy (May 2001).
2. R. Anderson, Trusted Computing FAQ – TCPA/Palladium/NGSCB/TCG, http://www.cl.cam.ac.uk/∼rja14/tcpa-faq.html.
3. R.J. Anderson, M.G. Kuhn, "Low Cost Attacks on Tamper-Resistant Devices", pp. 125–136, 5th International Workshop on Security Protocols, Springer LNCS 1361 (1997).
4. W.A. Arbaugh, D.J. Farber, J.M. Smith, "A Secure and Reliable Bootstrap Architecture", Proc. 1997 IEEE Symp. Security and Privacy, pp.65–71, May 1997.
5. W.A. Arbaugh, D.J. Farber, A.D. Keromytis, J.M. Smith, *Secure and Reliable Bootstrap Architecture*, U.S. Patent 6,185,678 (filed Oct.2 1998; issued Feb.6 2001).
6. D. Aucsmith, "Tamper Resistant Software: An Implementation", Proc. 1st International Information Hiding Workshop (IHW), Cambridge, U.K. 1996, Springer LNCS 1174, pp. 317-333 (1997).
7. D. Aucsmith, G. Graunke, *Tamper Resistant Methods and Apparatus*, U.S. Patent 5,892,899 (filed June 13 1996; issued Apr.6 1999).
8. B. Balacheff, L. Chen, S. Pearson (ed.), D. Plaquin, G. Proudler, *Trusted Computing Platforms: TCPA Technology in Context*, Prentice Hall, 2002.
9. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang, "On the (Im)possibility of Obfuscating Programs", pp. 1–18, Advances in Cryptology – Crypto 2001, Springer LNCS 2139 (2001).
10. E. Biham, A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems", pp. 513–525, Advances in Cryptology – Crypto '97, Springer LNCS 1294 (1997). *Revised*: Technion - C.S. Dept. - Technical Report CS0910-revised, 1997.
11. D. Boneh, R.A. DeMillo, R.J. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations", *J. Cryptology* 14(2), pp. 101–119 (2001).
12. H. Chang, M. Atallah, "Protecting Software Code by Guards", Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), Springer LNCS 2320, pp.160–175 (2002).
13. Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, M. Jakubowski, "Oblivious Hashing: A Stealthy Software Integrity Verification Primitive", Proc. 5th Information Hiding Workshop (IHW), Netherlands (October 2002), Springer LNCS 2578, pp.400–414.
14. D.M. Chess, "Security Issues in Mobile Code Systems", pp.1–14 in *Mobile Agents and Security*, G. Vigna (ed.), Springer LNCS 1419 (1998).
15. S. Chow, Y. Gu, H. Johnson, V.A. Zakharov, "An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs", pp. 144–155, Proc. ISC 2001 – 4th International Information Security Conference, Malaga, Spain 1–3 October 2001, Springer LNCS 2200 (2001).
16. J.R. Nickerson, S.T. Chow, H.J. Johnson, Y. Gu, "The Encoder Solution to Implementing Tamper Resistant Software", presented at the CERT/IEEE Information Survivability Workshop, Vancouver, Oct. 2001.
17. S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot, "White-Box Cryptography and an AES Implementation", Proc. 9th International Workshop on Selected Areas in Cryptography (SAC 2002), Springer LNCS 2595 (2003).
18. S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot, "A White-Box DES Implementation for DRM Applications", Proc. 2nd ACM Workshop on Digital Rights Management (DRM 2002), Springer LNCS (to appear).

19. F. Cohen, “Operating System Protection Through Program Evolution”, *Computers and Security* 12(6), 1 Oct. 1993, pp. 565–584.
20. C. Collberg, C. Thomborson, D. Low, “A Taxonomy of Obfuscating Transformations”, Technical Report 148, Dept. Computer Science, University of Auckland (July 1997).
21. C. Collberg, C. Thomborson, D. Low, “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs”, Proc. Symp. Principles of Programming Languages (POPL’98), Jan. 1998.
22. C. Collberg, C. Thomborson, D. Low, “Breaking Abstractions and Unstructuring Data Structures”, IEEE International Conf. Computer Languages (ICCL’98), May 1998.
23. C.S. Collberg, C. Thomborson, “Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection”, *IEEE Trans. Software Engineering*, Vol. 28 No. 6 (June 2002).
24. J. Daemen, V. Rijmen, *The Design of Rijndael:* AES *– The Advanced Encryption Standard*, Springer, 2001.
25. ComputerWeekly.com, “U.S. Software Security Takes Off”, 8 November 2002, http://www.computerweekly.com/Article117316.htm
26. P. England, J.D. DeTreville, B.W. Lampson, *Digital Rights Management Operating System*, U.S. Patent 6,330,670 (filed Jan.8 1999; issued Dec.11 2001).
27. P. England, J.D. DeTreville, B.W. Lampson, *Loading and Identifying a Digital Rights Management Operating System*, U.S. Patent 6,327,652 (filed Jan.8 1999; issued Dec.4 2001).
28. S. Forrest, A. Somayaji, D. H. Ackley, “Building Diverse Computer Systems”, pp. 67–72, Proc. 6th Workshop on Hot Topics in Operating Systems, IEEE Computer Society Press, 1997.
29. M.R. Garey, D.S. Johnson, *Computers and Intractability - A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.
30. O. Goldreich, R. Ostrovsky, “Software Protection and Simulation on Oblivious RAMs”, *Journal of the ACM*, vol.43 no.3 (May 1996), pp.431–473. Based on earlier ideas of Goldreich (STOC’87) and Ostrovsky (STOC’90).
31. J. Gosler, “Software Protection: Myth or Reality?”, Advances in Cryptology – CRYPTO’85, Springer-Verlag LNCS 218, pp.140–157 (1985).
32. P. Gutmann, “An Open-source Cryptographic Co-processor”, Proc. 2000 USENIX Security Symposium.
33. A. Herzberg, S.S. Pinter, “Public protection of software”, pp.371–393, *ACM Trans. Computer Systems*, vol.5 no.4 (Nov. 1987). Earlier version in Crypto’85.
34. B. Horne, L. Matheson, C. Sheehan, R. Tarjan, “Dynamic Self-Checking Techniques for Improved Tamper Resistance”, Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), Springer LNCS 2320, pp.141–159 (2002).
35. M. Jacob, D. Boneh, E. Felton, “Attacking an Obfuscated Cipher by Injecting Faults”, Proc. 2nd ACM Workshop on Digital Rights Management (DRM 2002), Springer LNCS (to appear).
36. M. Jakobsson, M.K. Reiter, “Discouraging Software Piracy Using Software Aging”, Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), Springer LNCS 2320, pp.1–12 (2002).
37. S. Kent, *Protecting Externally Supplied Software in Small Computers*, Ph.D. thesis, M.I.T., September 1980.
38. A. Kerckhoffs, “La Cryptographie Militaire”, *Journal des Sciences Militaires*, vol.9 (February 1883).

39. D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz, “Architectural Support for Copy and Tamper Resistant Software”, Proc. 9th International Conf. Architectural Support for Programming Languages and Operating Systems (Nov. 2000).
40. A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.
41. Next-Generation Secure Computing Base (formerly Palladium), Microsoft web site, http://www.microsoft.com/resources/ngscb/default.mspx.
42. Next-Generation Secure Computing Base - Technical FAQ, Microsoft web site, http://www.microsoft.com/technet/security/news/NGSCB.asp.
43. T. Ogiso, U. Sakabe, M. Soshi, A. Miyaji, “Software Tamper Resistance Based on the Difficulty of Interprocedural Analysis”, 3rd Workshop on Information Security Applications (WISA 2002), Korea, August 2002.
44. F. Petitcolas, R.J. Anderson, M.G. Kuhn, “Information Hiding – A Survey”, *Proc. of the IEEE* (Special Issue on Protection of Multimedia Content), vol.87 no.7 (July 1999), pp.1062–1078.
45. T. Sander, C.F. Tschudin, “Towards Mobile Cryptography”, pp. 215–224, Proc. 1998 IEEE Symposium on Security and Privacy.
46. T. Sander, C.F. Tschudin, “Protecting Mobile Agents Against Malicious Hosts”, pp. 44–60 in *Mobile Agents and Security*, G. Vigna (ed.), Springer LNCS 1419 (1998).
47. F. Schneider (ed.), *Trust in Cyberspace*, report of the Committee on Information Systems Trustworthiness, Computer Science and Telecommunications Board, (U.S.) National Research Council (National Academy Press, 1999).
48. Trusted Computing Group, http://www.trustedcomputinggroup.org/home.
49. N. van Someren, A. Shamir, “Playing Hide and Seek with Keys”, pp. 118–124, Financial Cryptography’99, Springer LNCS 1648 (1999).
50. J. Wang, “Average-Case Computational Complexity Theory”, pp.295–328 in: *Complexity Theory Retrospective II*, L. Hemaspaandra and A. Selman (eds.), Springer, 1997.
51. C. Wang, J. Hill, J. Knight, J. Davidson, “Software Tamper Resistance: Obstructing Static Analysis of Programs”, Dept. of Computer Science, Univ. of Virginia, Tech. Report CS-2000-12 (May 2000). Updated in [52].
52. C. Wang, *A Security Architecture for Survivability Mechanisms*, Ph.D. thesis, University of Virginia (Oct. 2000).