CERIAS Tech Report 2004-55

A SURVEY OF ANTI-TAMPER TECHNOLOGIES

by Eric D. Bryant, Mikhail J. Atallah, Martin R. Stytz

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

A Survey of Anti-Tamper Technologies

Dr. Mikhail J. Atallah, Eric D. Bryant, and Dr. Martin R. Stytz
Arxan Technologies, Inc.

This article surveys the various anti-tamper (AT) technologies used to protect software. The primary objective of AT techniques is to protect critical program information by preventing unauthorized modification and use of software. This protection goal applies to any program that requires protection from unauthorized disclosure or inadvertent transfer of leading-edge technologies and sensitive data or systems. In this article, we review the various approaches to AT techniques, their strengths and weaknesses, their advantages and disadvantages, and briefly discuss a process for developing program protection plans. We also survey the tools that are typically used to circumvent AT protections, and techniques that are commonly used to make these protections more resilient against such attack.

The unauthorized modification and subsequent misuse of software is often referred to as software *cracking*. Usually, cracking requires disabling one or more software features that enforce policies (of access, usage, dissemination, etc.) related to the software. Because there is value and/or notoriety to be gained by accessing valuable software capabilities, cracking continues to be common and is a growing problem.

To combat cracking, anti-tamper (AT) technologies have been developed to protect valuable software. Both hardware and software AT technologies aim to make software more resistant against attack and protect critical program elements. However, before discussing the various AT technologies, we need to know the adversary's goals. What do software crackers hope to achieve? Their purposes vary, and typically include one or more of the following:

- **Gaining unauthorized access**. The attacker's goal is to disable the software access control mechanisms built into the software. After doing so, the attacker can make and distribute illegal copies whose copy protection or usage control mechanisms have been disabled – this is the familiar software piracy problem. If the cracked software provides access to classified data, then the attacker's real goal is not the software itself, but the data that is accessible through the software. The attacker sometimes aims at modifying or *unlocking* specific functionality in the program, e.g., a demo or *export* version of software is often a deliberately degraded version of what is otherwise fully functional software. The attacker then seeks to make it fully functional by re-enabling the missing features.
- **Reverse engineering.** The attacker aims to *understand* enough about the software to steal key routines, to gain access to proprietary intellectual property, or to carry out *code-lifting*, which consists of reusing a crucial part of the code (without necessarily understanding the internals of how it works) in some other software. Good programming practices, while they facilitate software engineering, also tend to simultaneously make it easier to carry out reverse engineering attacks. These attacks are potentially very costly to the original software developer as they allow a competitor (or an enemy) to nullify the developer's competitive advantage by rapidly closing a technology gap through insights gleaned from examining the software.
- **Violating code integrity.** This familiar attack consists of either injecting malicious code (*malware*) into a program, injecting code that is not malevolent but illegally enhances a program's functionality, or otherwise subverting a program so it performs new and unadvertised functions (functions that the owner or user would not approve of). While AT technology is related to anti-virus protection, it has some crucial differences. AT technology is similar to virus protection in that it impedes malware infection of an AT-protected executable. However, AT technology differs from virus protection in that the AT technology's goal is not only to protect the client's software from unauthorized modification by malevolent outsiders (infection by malware written by others), but also to protect the software from modification by an authorized client. In many situations, it is important that only authorized applications execute (e.g., in a taximeter, odometer, or any situation where tampering is feared), using only authorized functionality, and that only valid data is used.

It should be clear by now that AT technology is not only about anti-piracy, it has an equal and broader aim of policy enforcement. That aim is to enforce the policies of the software publisher about the proper use of the software, even as the software is running in a potentially hostile environment where the user *owns the processor* and is intent on violating those policies.

There is a plethora of AT protection mechanisms. These include encryption wrappers, code obfuscation, guarding, and watermarking/fingerprinting in addition to various hardware techniques. While these techniques are discussed separately for pedagogical purposes, the reader should bear in mind that software is best protected when several protection techniques are used together in a mutually supportive manner. No technique is invulnerable or even clearly superior to the others in all circumstances; therefore, a mix of protection techniques allows the defense to capitalize on the strengths of each technique while also masking the shortfalls of other techniques. In the following paragraphs we present a brief overview of these techniques.

Hardware-Based Protections

The most common hardware approach uses a trusted processor. The trusted, tamper-resistant hardware checks and verifies every piece of hardware and software that exists – or that requests to be run on a computer – starting at the boot-up process [1]. This hardware could guarantee integrity by checking every entity when the machine boots up, and every entity that will be run or used on that machine after it boots up. The hardware could, for example, store all of the keys necessary to verify digital signatures, decrypt licenses, decrypt software before running it, and encrypt messages during

any online protocols it may need to run (e.g., for updates) with another trusted remote entity (such as the software publisher).

Software downloaded onto a machine would be stored in encrypted form on the hard drive and would be decrypted and executed by the hardware, which would also encrypt and decrypt information it sends and receives from its random access memory. The same software or media could be encrypted in a different way for each trusted processor that would execute it because each processor would have a distinctive decryption key. This would put quite a dent in the piracy problem, as disseminating your software or media files to others would not do them much good (because their own hardware would have different keys).

A less drastic protection than using a separate, trusted, hardware computational device also involves hardware, but is more lightweight such as a smart card or physically secure *token*. These lightweight hardware protection techniques usually require that the hardware be present for the software to run, to have certain functionality, to access a media file, etc. Defeating this kind of protection usually requires *working around* the need for the hardware rather than duplicating the hardware. The difficulty of this work-around depends on the role that the tamper-resistant hardware plays in the protection. A device that just outputs a serial number is trivially vulnerable to a *replay attack* (e.g., an attacker replays a valid serial number to the software, without the presence of the hardware device), whereas a smart card that engages in a challenge-response protocol (different data each time) prevents the simple replay attack but is still vulnerable (e.g., to modification of the software interacting with the smart card). A device that decrypts content or that provides some essential feature of a program or media file is even harder to defeat.

Advantages and Drawbacks

The chief advantage of hardware-based protection techniques is that they run on a trusted CPU and can be made arbitrarily complex – hence, difficult to defeat while inflicting minimal computational cost on the protected software once it has been decrypted within the hardware and is running. However, there is a cost to decrypt it in the first place, and also to encrypt everything that goes out to the non-protected part of the system, and then decrypt it when it comes back into the trusted hardware.

In addition, it is generally more difficult to successfully attack tamper-resistant hardware and make the exploit directly available to others than a software-only protection scheme. This point holds only for a properly designed system. A compromise of hardware that imprudently contains the same secret keys as all other hardware of the same type would lead to widely reproducible exploits.

The advantages of hardware protection also include its capability to enforce such rules as "only approved peripherals can be a part of this computer system," or "only approved (through digital signatures) software and contents are allowed," etc.

Nevertheless, hardware-based protection also has its drawbacks. There is the usual problem of inflexibility: hardware-based protections are more awkward to modify, port, and update than software-based ones. They are also less secure than commonly assumed and can be broken; see, e.g., [2]. To date, it has not been demonstrated that hardware protections can scale to grid computing or to small-scale computing. In addition, there is no guarantee that all avenues of attack are closed by hardware protection, and there is a significant cost attached to using hardware protection; the cost is driven mainly by the time needed to assemble, integrate, and test the hardware protection technique.

"No technique is invulnerable or even clearly superior to the others in all circumstances; therefore, a mix of protection techniques allows the defense to capitalize on the strengths of each technique while also masking the shortfalls of other techniques."

Additional drawbacks to the hardware protection approach include its expense and general *fragility* to accidents (an electric power surge that *fries* the processor also renders the hard drive contents unusable because the key that decrypts them is destroyed). The potential implications for censorship are also chilling. Another disadvantage of hardware protection is the boot-up time and the time spent encrypting and decrypting, which makes the approach problematic for low-end machines and embedded systems (unless the whole system lies within tamper-resistant hardware).

Using trusted hardware also incurs many indirect costs as a result of the earlier-mentioned limitations it imposes (e.g., the restriction to only certain *approved* hardware, software, and media creates a barrier to competition that leads to higher prices). Due to the imperfect protection offered by hardware protection, a more robust approach to software security interweaves hardware protection with other protection techniques such as those discussed in the following sections.

The rest of this article discusses the various software-based protection mechanisms. The reader should keep in mind that hardware and software protection techniques are not mutually exclusive. A judicious combination can serve to increase the security of the system more than any of its individual component techniques.

Encryption Wrappers

With encryption wrapper software security, critical portions of the software (or possibly all of it) are encrypted and decrypted dynamically at run-time. The encryption wrapper approach works well against a static attack, and forces the attacker to run the program in order to get an unencrypted image of it. To make the attacker's task harder, at no time during execution is the whole software *in the clear*; code decrypts just before it executes, leaving other parts of the program still encrypted. Therefore, no single *snapshot* of memory can expose the whole decrypted program. Of course, the attacker can take many such snapshots, compare them, and piece together the unencrypted program.

Another avenue of attack is to figure out the various decryption keys that are present in the software. One defensive technique that can be used to delay the attacker is to include defensive mechanisms in the program that deprive the attacker of using run-time attack tools, e.g., anti-debugger, anti-memory dump, and other defensive mechanisms, which make it more difficult for the attacker to run and analyze the program in a synthetic (virtual machine) environment. Yet, a determined attacker can usually defeat

these protections (e.g., through the use of virtual machines that faithfully emulate a PC, including the most rarely used instructions, cache behavior, etc).

Encryption wrappers often use *lightweight* encryption to minimize the computational cost of executing the protected program. The encryption can be advantageously combined with compression: Not only does this result in a smaller amount of storage usage, but it also makes the encryption harder to defeat by cryptanalysis (of course one compresses before encryption, not the other way around).

An encryption wrapper's chief advantage is that it effectively hinders an attacker's ability to statically analyze a program. The attacker is then forced to perform more sophisticated types of dynamic attacks, which can significantly increase the amount of time needed to defeat the protection. The main disadvantage of encryption wrappers is the performance penalty caused by the decryption overhead, and its weakness to memory dumps: before it can run, encryption-protected software must be decrypted, at which point it becomes exposed.

Code Obfuscation

Code obfuscation consists of transforming code so it becomes less intelligible to a human, thus making it not only harder to reverse engineer, but also harder to tamper with. In software that has specific areas where policy checks are made, these areas will be harder to identify and disable after the software has been obfuscated. Obfuscation is usually carried out by inserting or performing obfuscating transformations. It is a requirement that these transformations do not damage a program's functionality, and it must have only a moderate impact on code performance, and on the storage space used on the disk and at run-time (of the two, speed is more important).

The obfuscation must also be resilient to attack, and for this reason it is desirable to maximize the *obscurity* of the obfuscated software. The obfuscating transformations need to be resilient against tools designed to automatically undo them, and to not be easily detectable by statistical analysis of the resulting code (resilience to statistical analysis makes it harder for automatic tools to find the locations where these transformations were applied).

The different types of obfuscation transformations that have been proposed [3] include the following:

- **Layout obfuscation.** This modifies the *physical appearance* of the code, e.g., replacing important variables with random strings, removing all formatting (making nested conditional statements harder to read), etc. Such transformations are easy to make but are effective only when combined with other transformation techniques.
- **Data obfuscation.** This obscures the data structures used within a program, e.g., the representation and the methods of using that data, independent data merging (and vice-versa – splitting up data that is dependent), etc. Data obfuscation serves to delay the attacker because data structures contain important information that any attacker needs to comprehend before launching an attack.
- **Control obfuscation.** This manipulates the control flow of a program to make it difficult to discern its original structure, e.g., through merging (or splitting) various fragments of code, reordering expressions, loops, or blocks, etc. It is similar to creating a spurious program that is *entangled* with the original program so as to obscure the important control features of that program.
- **Preventive transformations.** These aim at making it difficult for a de-obfuscation tool to extract the *true* program from the obfuscated version of it. Preventive transformations can be implemented by using what Collberg [4] calls opaque predicates, an example of which is a conditional statement that always evaluates as true, but in a manner that is hard to recognize.

Obfuscation can be done at the source-code level (source-to-source translation) or at the assembly level. Although most obfuscators are of the former kind (source-to-source), assembly level obfuscation is better because it effectively hides the operation of the binary. If the source-code level transformations hide information by adding crude and inefficient ways of doing simple tasks, then the code optimizer in the compiler may undo them. If, on the other hand, the transformations are clever enough to fool the optimizer, then it can fail to properly do its job, and the performance of the resulting code suffers. Low-level obfuscation does not prevent the code optimizer from doing its job, but if done carelessly it runs the risk of producing code that looks so different from the kind produced by the compiler that it inadvertently *flags* the areas where the transformations were applied.

Obfuscation transformations are classified according to several criteria: how much obscurity they add to the program (potency), how difficult they are to break for a de-obfuscator (resilience), and how much computational overhead they add to the obfuscated application (cost). In [4], software complexity metrics are used to formalize the notion of transformation potency and resilience.

The potency of a transformation measures how much more difficult the obfuscated code is to understand for a human than the original code. On the other hand, the resilience of a transformation measures how well it stands up to attack by an automatic de-obfuscator. The resilience measurement takes two factors into account: the programmer effort required to construct the de-obfuscator and the execution time and space required by the de-obfuscator to reduce the potency of the transformation. The best obfuscation is usually achieved by a combination of the above three mentioned transformations. The combination of the three approaches provides a well-balanced mix of highly potent and resilient transformations.

Like all software-only protections, obfuscation can delay – but not prevent – a determined attacker intent on reverse-engineering the software. Barak [5] presents a family of functions that are provably impossible to completely and successfully obfuscate. For more information and a discussion of code obfuscation, refer to [3, 4, 6, 7].

Software Watermarking and Fingerprinting

The goal of watermarking is to embed information into software in a manner that makes it hard to remove by an adversary without damaging the software's functionality. The information inserted could be purchaser information, or it could be an integrity check to detect modification, the placing of caption-type information, etc. A watermark need not be stealthy; visible watermarks act as a deterrent (against piracy, for example), but most of the literature has focused on stealthy watermarks. In steganography (the art of concealing the existence of information within seemingly innocuous carriers), the mark is required to be stealthy: its very existence must not be detectable [8].

A specific type of watermarking is fingerprinting, which embeds a unique message in each instance of the software for traitor tracing. This has consequences for the adversary's ability to attack the

watermark: two differently marked copies often make possible a *diff* attack that compares the two differently marked copies and can enable the adversary to create a usable copy that has neither one of the two marks. Thus, in any fingerprinting scheme, it is critical to use techniques that are resilient against such comparison attacks.

A watermark is generally required to be robust (hard to remove). In some situations, however, a fragile watermark is desirable; it is destroyed if even a small alteration is made to the software (e.g., this is useful for making the software tamper-evident).

Software watermarks can be static, i.e., readable without running the software, or could appear only at run-time (preferably in an evanescent form). In either case, reading the watermark usually requires knowing a secret key, without which the watermark remains invisible.

Watermarks may be used for proof of software authorship or ownership, fingerprinting for identifying the source of illegal information/software dissemination, proof of authenticity, tamper-resistant copyright protection, and captioning to provide information about the software. When software watermarks are used for proof of authorship or ownership (culprit-tracing), it is important to use a very resilient scheme. Recall that this is when the watermark contains information about the copyright owner as well as the entity that is licensed to use the software, thus allowing trace-back to the culprit if the item were to be illegally disseminated to others. Breaking the security of such a scheme can enable the attacker to *frame* an innocent victim.

As you can see, while watermarks can demonstrate authorized possession and the fact that software has been pirated, they do not address the reverse engineering or authorized execution issues of the other schemes discussed; therefore, we advocate the development and use of a spectrum of software protection techniques.

Guarding

A guard is code that is injected into the software for the sake of AT protection. A guard must not interfere with the program's basic functionality unless that program is tampered with – it is the tampering that triggers a guard to take action that deviates from normal program behavior. Examples of guard functionality range from tasks as simple as comparing a checksum of a code fragment to its expected value, to repairing code (in case it was maliciously damaged), to complex and indirect forms of protection through subtle side effects.

The preferred use of the guarding approach consists of injecting into the code to be protected a large number of guards that mutually protect each other as well as the software program in which they now reside. Guards can also be used to good effect in conjunction with hardware-based protection techniques to further ensure that the protected software is only executed in an authorized environment.

The number, types, and stealthiness of guards; the protection topology (*who protects who*); and where the guards are injected in the original code and how they are entangled with it are some of the parameters in the strength of the resulting protection: They are all tunable in a manner that depends on the type of code being protected, the desired level of protection, etc.

Manually installing such a tangled web of protection is impractical (as it must be done every time the software is updated), so it is important that this protection be done in a highly automated fashion using high-level scripts that specify the protection guidelines and parameters. It should be thought of as a part of the compilation process where an *anti-tamper* option results in code that is guarded and tamper-resistant.

The rationale for this approach is that a single (even if elaborate) AT protection scheme for a software application is insufficient because a single defense results in a single point of attack that can be located and compromised. To make self-protection robust, the defense must not rely on a single complex protection technique no matter how effective it might be. Instead, there needs to be a multitude of (possibly simple) protection techniques installed in the program that cooperatively enforce the code's integrity as well as protect the other against tampering.

The guard's response when it detects tampering is flexible and can range from a mild response to the disruption of normal program execution through injection of run-time errors (crashes or even subtle errors in the answers computed); the reaction chosen depends on the software publisher's business model and the expected adversary. Generally, it is better for a guard's reaction to be delayed rather than to occur immediately upon detection so that tracing the reaction back to its true cause is as difficult as possible and consumes a great deal of the attacker's time. More on guarding can be found in [9].

AT Process

This section explores the various AT guidelines expressed in the "Defense Acquisition Guidebook" [10], and the recommended process for developing a program protection plan. The "Defense Acquisition Guidebook" specifies the AT measures that should be considered for use on any system with critical program information (CPI), developed with allied partners, likely to be sold or provided to United States allies and friendly foreign governments, or likely to fall into enemy hands. The first step in the recommended AT methodology is to develop a program protection plan. The process of developing this plan includes the following:

- Develop a list of critical technologies.
- Develop a threat analysis.
- Develop a list of identified vulnerabilities.
- Develop a preliminary AT requirement.
- Perform an analysis of AT methods that applies to the system, including cost/benefit assessments.
- Provide an explanation of which AT method(s) will be implemented; develop a plan for validating the AT implementation.

The standard approach of validating AT protections is done via *red-teaming*. A red team consists of individuals who are well versed in security methods and their corresponding weaknesses. Their primary objectives are to attempt to defeat the protection, to assess the protection's strengths and weaknesses, and to make recommendations for improvement. While this is an effective method of evaluation, a major problem with red teams is that the validation is done by humans, and may not be totally reliable or repeatable. Furthermore, as the need for AT technologies grows, red teams are becoming increasingly called upon to perform evaluations. The teams are overwhelmed with assignments, significant delays in product evaluations, and release results. To improve this process, there is a clear and present need for automated testing and validation tools. Such tools could be used to define a standard set of metrics and guidelines to evaluate software protections.

Conclusion

This article has surveyed the motivation for using AT technology, the hardware and software AT techniques in use today, and the strengths and weaknesses of AT technologies. We also briefly introduced

the process and documentation used to develop a program protection plan. The motivation for and primary objective of AT technology is to protect CPI by preventing unauthorized modification and use of software. The main software AT techniques include encryption wrappers, code obfuscation, watermarking/fingerprinting, and guarding.

A fundamental challenge faced by software AT technology is that the protected application is running on a host that is not trusted, and thus cannot be assured to be secure. Guards address this shortfall to a degree and in a flexible and extensible manner. However, in light of the need for robust, seamless, comprehensive software defense, using both software and hardware AT solutions in combination often offers an appealing alternative to using them individually (especially if economic considerations are factored in).

At this time, indications are that if strong software AT technology (e.g., in the form of judiciously constructed guards) is added to an application so that it requires the presence of a *lightweight* tamper-resistant hardware device in order to execute properly, the result is a strong yet economical software protection capability.◆

References

1. Arbaugh, W., D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. Proc. of the IEEE Symposium on Security and Privacy, Oakland, CA, 1997.
2. Anderson, R., and M. Kuhn. Tamper Resistance – A Cautionary Note. Proc. of Second Usenix Workshop on Electronic Commerce, Oakland, CA, Nov. 1996: 1-11.
3. Collberg C., and C. Thomborson. "Watermarking, Tamper-Proofing, and Obfuscation Tools for Software Protection." IEEE Transactions on Software Engineering 28.8 (2002): 735-746, 2002.
4. Collberg, C., C. Thomborson, and D. Low. "A Taxonomy of Obfuscating Transformations." Department of Computer Science, University of Auckland, New Zealand, 1997.
5. Barak, B., et al. "On the (Im)possibility of Obfuscating Programs." Electronic Colloquium on Computational Complexity. Report No. 57, 2001.
6. Wang, C., et al. "Software Tamper Resistance: Obstructing Static Analysis of Programs." University of Virginia, Computer Science Technical Report CS-2000-12, Dec. 2000.
7. Wroblewski, G. "General Method of Program Code Obfuscation." Diss. Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.
8. Johnson, N. "Introduction to Steganography and Steganalysis." Workshop on Statistical and Machine Learning Techniques in Computer Intrusion Detection, Johns Hopkins University, 11-13 June 2002.
9. Chang, H., and M. Atallah. Protecting Software Code By Guards. Proc. of ACM Workshop on Security and Privacy in Digital Rights Management, Philadelphia, PA, Nov. 2001: 160-175.
10. Office of the Secretary of Defense. Interim Defense Acquisition Guidebook. Washington, D.C.: OSD, 30 Oct. 2002 <http://dod5000.dau.mil/DoD5000Interactive/InterimGuidebook.asp>.

Additional Reading

1. Lipton, R.J., S. Rajagopalan, and D.N. Serpanos. "Spy: A Method to Secure Clients for Network Services." IEEE Distributed Computing Systems Workshops 2002: 23-28.
2. Anderson, R., and M. Kuhn. "Low Cost Attacks on Tamper Resistant Devices." 5th International Workshop on Security Protocols, Apr. 1997: 125-136.

About the Authors

Mikhail "Mike" J. Atallah, Ph.D., is a distinguished professor in the Computer Science Department at Purdue University. His main research interests are in information security. A fellow of the Institute of Electrical and Electronics Engineers, Atallah has been both a keynote and invited speaker at many national and international meetings, and a speaker in the Distinguished Colloquium Series of many top computer science departments. He is a co-founder of Arxan Technologies Inc. Atallah has a Master of Science and a doctorate degree from Johns Hopkins.

Arxan Technologies, Inc.
3000 Kent AVE
STE D2-100
West Lafayette, IN 47906
Phone: (765) 494-6017 ext.54
Fax: (765) 496-3181
E-mail: mja@cs.purdue.edu

Eric D. Bryant is a research engineer for Arxan Technologies, Inc., and is pursuing his doctorate degree at Purdue University. His primary research interests are in information security, reverse engineering, compiler and programming language design, and artificial intelligence. Bryant has a Bachelor of Science in computer science from Purdue University.

Arxan Technologies, Inc.
3000 Kent AVE
STE D2-100
West Lafayette, IN 47906
Phone: (765) 775-1004 ext. 106
Fax: (765) 775-1004
E-mail: ebryant@arxan.com

Martin R. Stytz, Ph.D., is a senior research scientist and engineer for Calculated Insight and formerly for the Air Force Research Laboratory at Wright-Patterson Air Force Base, Ohio. He is a member of the Institute of Electrical and Electronics Engineers (IEEE) Task Force on Security and Privacy, and is on the editorial board for IEEE's Security and Privacy. Stytz has a Bachelor of Science from the U.S. Air Force Academy, a Master of Arts from Central Missouri State University, and a Master of Science and doctorate degree from the University of Michigan.

Arxan Technologies, Inc.
3000 Kent AVE
STE D2-100
West Lafayette, IN 47906
Phone: (407) 497-4407
Fax: (765) 775-1004
E-mail: mstytz@att.net