Towards Efficient Software Protection Obeying Kerckhoffs's Principle using Tamper-proof Hardware

Brandon Broadnax[1], Matthias Huber[2], Bernhard Löwe[3],
Jörn Müller-Quade[1] and Patrik Scheidecker[2]
[1]*Karlsruhe Institute of Technology, Karlsruhe, Germany*
[2]*FZI Research Center for Information Technology, Karlsruhe, Germany*
[3]*Rohde & Schwarz GmbH & Co. KG, Cologne, Germany*

Keywords: Software Protection, Kerckhoffs's Principle, Tamper-Proof Hardware.

Abstract: We propose the first software protection scheme obeying Kerckhoffs's principle that is suited for practical implementation. Previous schemes have either been closed source or too inefficient to be considered practically viable. A key technique of our scheme is to partition the software in such a way that a hacker who knows a set of parts cannot learn additional ones. To achieve a partition with this property, our scheme exploits the domain knowledge that is necessary to create the software as well as the inherent complexity of the software's code. If a software is sufficiently complex to admit such a partition then we can prove that there are no successful attack strategies on our scheme other than storing every line of code that has been executed.

1 INTRODUCTION

Software has become an increasingly important commercial factor. Industrial machinery, cars or smart devices contain more and more software innovations. As a consequence, software protection has become an important field of IT security. The goal of software protection is to make it as hard as possible for a hacker to construct an unauthorized copy of a software.

Conventional software protection methods such as requiring serial numbers or software activation codes are not sufficient to prevent piracy. Modern software protection schemes use cryptographic methods such as encryption schemes or digital signatures. The secret keys are stored in a tamper-proof hardware, a so-called *dongle*, which is distributed with the software. The software therefore only runs properly if the dongle is attached to the electronic device in use. A hacker who wants to illegally resell the software has to create a copy that runs without a dongle. Since a hacker may also have access to one or several dongle(s), one cannot prevent him from running the program and storing the parts of the program code that were executed (memory dump).

Unfortunately, all protection schemes that are used in practice rely on keeping their mechanisms secret. Thus, it is relatively easy for hackers who know these mechanisms sufficiently well to break such protection schemes. These schemes thus violate Kerckhoffs's principle which requires that a scheme should be secure even if everything about it is public knowledge, except for cryptographic keys.

Kerckhoffs's principle, put forward by Auguste Kerckhoffs in 1883, offers many advantages. For instance, since cryptographic keys can be chosen independently for each program, compromized keys have no impact on another protected program and new keys can be chosen without having to modify the scheme. Moreover, since the mechanisms of the protection scheme do not have to be kept secret, they are open to public scrutiny.

In theory, there are methods that can be used to provably guarantee software protection in accordance with Kerckhoffs's principle , e.g. (Goldreich and Ostrovsky, 1996; Lin, 2016). These methods, however, have been useless in practice so far due to the high computational overhead of the cryptographic primitives used in these methods.

We propose the first software protection scheme that obeys Kerckhoffs's principle and is also suited for practical implementation. At the core of this scheme lies a simple assumption: a hacker, although perhaps skilled at attacking cryptographic mechanisms, is lacking the domain knowledge that is necessary to create the software product he seeks to copy illegally. For instance, a hacker may not be familiar with the un-

DOI: 10.5220/0006906105530558
In *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications (ICETE 2018) - Volume 2: SECRYPT*, pages 553-558
ISBN: 978-989-758-319-3

derlying mathematics of a scientific program such as a computer algebra system. This assumption is supported by the fact that if a hacker knew the relevant domain knowledge then he could write the software himself without having to attack the protection at all. Hence, this is a necessary assumption for every software protection scheme.

This lack of domain knowledge can be exploited to obtain a secure protection. The main idea is to *partition* the program code of a software in such a way that, given an arbitrary set of parts, it is hard to come up with the remaining parts if one lacks the domain knowledge that is necessary to create the software. Of course, not all programs admit a partition with such a security property. A program needs to have "sufficient complexity" for such a partition to be possible.

Identifying which code structures are suitable for such a partitioning is part of our current research. We give a brief overview of our current state of research in section 5.

2 RELATED WORK

An effective software protection scheme has to reach at least two goals. The first one is obvious: to prevent an adversary from creating a copy of the program that runs without a valid license. Since an adversary can always create an exact copy of the software, the execution of the software has to be tied to a physical device. The second goal is closely related to the first one. The adversary must not learn how the program works. If he knows the "inner workings" of the protected program then he can write a functionally equivalent program by himself. For this reason, software protection and *code obfuscation* are often mentioned in the same breath.

Code obfuscation is the process of modifying machine code in such a way that it becomes unintelligible, while leaving the functionality of the original program code unaltered. Various types of code obfuscation can be found in the literature. *Black-Box Obfuscation* (Barak et al., 2001) guarantees that every information that can be derived from the obfuscated program can also be learned given black-box access to the original program. However, it has been shown (Barak et al., 2001) that black-box obfuscation for arbitrary programs is impossible (without hardware assumptions). Only a few applications of black-box obfuscation are known, e.g. point-functions (Wee, 2005).

Another type of obfuscation is *indistinguishability obfuscation*. Indistinguishability obfuscation guarantees that the obfuscated code of two programs having the same functionality are indistinguishable. Some candidate indistingushiability obfuscators for arbitrary programs have been constructed, e.g. (Garg et al., 2016; Lin, 2016; Lin and Tessaro, 2017). Unfortunately, these constructions are impractical since they use cryptographic methods (e.g. fully-homomorphic encryption schemes) that are so far too inefficient for practical purposes.

Apart from merely modifiying the program code, other obfuscation techniques using *tamper-proof hardware tokens* have been explored. Since these hardware tokens can be implemented by dongles, they are also of interest for software protection. Hardware tokens (Katz, 2007) allow black-box obfuscation for all practically relevant functionalities. However, these constructions are not suited for practical implementation because they treat programs as circuits.

Another technique that has been explored in the context of software protection is *oblivious RAM* (Goldreich and Ostrovsky, 1996; Pinkas and Reinman, 2010). Oblivious RAM is a method for hiding the memory access pattern. Oblivious RAM can be combined with a physically shielded CPU to construct software protection schemes. Since we assume that the adversary has full control over the CPU, these constructions are not applicable in our setting. The SGX-technology of intel may change this in the future.

Moreover, white-box cryptography provides another type of obfuscation (Joye, 2008). The goal of white-box cryptography is not to hide what the program code does but rather to hide the key of the cryptographic operations (encryption, decryption, etc.) within the program code. However, known white-box cryptography solutions, e.g. (Karroumi, 2010), do not prevent unauthorized copying of the software but only protect the contents the software operates with.

3 OUR CONSTRUCTION

Our scheme uses an external tamper-proof hardware (dongle). This device is able to store small amounts of data, execute some code and decrypt ciphers. In the following, we will explain the mechanisms of our scheme in detail.

3.1 Encryption

We use a secret key encryption scheme as a basic building block in our construction. For performance reasons, each plaintext m is encrypted with a key k which in turn is encrypted with the secret key sk of the dongle. Decryption is carried out by having the dongle decrypt the key k which can then be used to decrypt the ciphertext generated with k on the (fast) CPU of the computer.

3.2 Partitioning a Program

Encrypting the program as a whole with one key would reveal it entirely after decryption. We therefore partition the program into several *program blocks* before encryption. Each program block is encrypted with a different key. These keys are encrypted together with the variant ID using the secret key stored in the dongle (see figure 1). Program blocks can therefore only be decrypted with the help of the dongle. Note that a program block is only decrypted when needed. Since every code block is encrypted we add an initial bootstrapping block that sends the first encrypted key to the dongle.

3.3 Duplication and Modification of Program Blocks

Given a program block, its input range is partitioned into several segments. For each segment a copy of the program block is created. Each copy is modified such that it still yields correct results within the segment of the input range it was created for, but yields false results for any input value outside this segment. We call this copy a *variant* of the program block. Furthermore, we introduce *wrapper functions* that determine the index of the next variant based on the current input. Each wrapper function is encrypted with the key of the dongle (see figure 2).

The creation of variants is an important part of our protection scheme. In order to achieve a meaningful notion of security, the set of variants must fulfill the following requirement:

Variant Security Assumption (VSA). *Given a subset of variants $\{\mathcal{P}_{i,j}\}$, it is hard for an adversary to come up with* additional *variants.*

A program has to be sufficiently complex to allow for creating variants that meet this requirement. In Section 5, we will sketch some techniques that are useful for creating variants that fulfill the VSA.

3.4 Traps

Traps are a special class of variants that are added to the program code (see figure 2). Traps do not belong to the original program and are never called during regular executions of the program. If an adversary tries to decrypt a trap, the dongle *locks* itself, invalidating the license. Traps prevent the adversary from simply decrypting one variant after the other using the dongle.

3.5 Evaluation of the Wrapper Functions on the Dongle

Code moving is the method of moving the execution of program code into the dongle. In theory, this technique provides a nearly perfect solution for software protection. However, it is inefficient in many cases because the dongle is a constrained hardware. Due to these performance issues, code moving cannot be used extensively. We use code moving only for the determination of the next program block, i.e., for the wrapper functions. This prevents an adversary from learning which code block is a trap by analyzing the wrapper functions.

3.6 Using the Dongle's State Memory

Since the dongle knows which variant will be executed next (because it calculated the index of the next variant, see Section 3.5), it expects that variant to be decrypted next. Every other variant that is sent to the dongle in the next decryption query will force the dongle to lock itself. We can realize this technique with only a small amount of memory in the dongle. Note that this technique also prevents illegal parallel executions of the protected program (e.g. via a cloud server).

3.7 Applying a Message Authentication Code

In order to prevent an adversary from manipulating the encrypted keys, variant IDs or wrapper functions, we apply a message authentication code (MAC) to the respective ciphertexts. The key of the MAC is stored in the dongle.

4 SECURITY

In this section, we give a brief informal security analysis of our scheme.

In the field of software protection, one generally distinguishes the following two types of adversaries:

Static Adversaries
Static adversaries pursue a strategy that is *independent* of the program code, using the dongle as a decryption oracle.

Dynamic Adversaries
Dynamic adversaries can run the program and analyze the structure of the program code (using, e.g., a debugger).

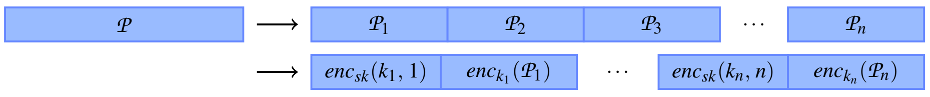

Figure 1: A program $\mathcal{P}$ is partitioned into n program blocks. Each block is encrypted separately.

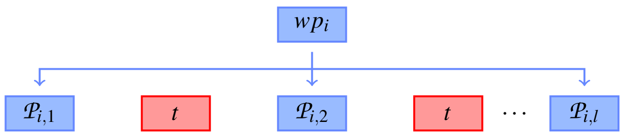

Figure 2: Traps are added. Wrapper functions determine the next variant.

We can prove that our scheme is secure against the above-mentioned adversary types given the following assumptions:

- **Secure Encryption:** The encryption scheme is IND-CPA-secure[1]
- **Secure MAC:** The message authentication code is EUF-CMA-secure.[1]
- **VSA:** The variant security assumption holds for the given program.
- **Tamper-Proof Hardware:** The key stored in the dongle is hard to extract.

In order to prove security, we have defined a mathematical model tailored to our setting. In the following, we briefly sketch our model as well as some of the arguments in the security proof.

Let us first consider static adversaries. First note that an (arbitrary, even non-static) adversary cannot send arbitrary code to the decryption oracle (dongle) since this would require forging the EUF-CMA-secure MAC because the dongle checks if the MAC tag of a received message is valid. Therefore, an adversary can only send encrypted variants to the oracle (more specifically, the ciphertexts containing the keys and variant IDs, see section 3.2). However, since the encrypted program code contains traps, a static adversary will fail with high probability (depending on the number of traps) at retrieving all variants by using the decryption oracle. Assuming the encryption scheme is IND-CPA-secure, a static adversary who has invalidated the decryption oracle by decrypting a trap cannot learn additional variants by attacking the encryption scheme.

Now let us discuss dynamic adversaries. Consider the attack strategy that simply runs the program and stores all the variants that have been decrypted during the program execution (memory dump). This type of attack is called a *Copy-and-Paste attack*. This attack cannot be prevented by our scheme. However, if the protected program is sufficiently large, then this attack is impractical since an adversary would have to run the program on an impractically large set of input values in order to retrieve the entire program.

Still, the fact that there exists an attack that is unpreventable makes defining security difficult, as a straightforward statement such as “no (dynamic) adversary can break the scheme” cannot be achieved. However, one can hope that there is no attack strategy that is “better” than the Copy-and-Paste strategy.

We have therefore formalized security using the “Real/Ideal-Paradigm” (Canetti, 2000). The “real model” captures all possible actions of dynamic adversaries given the encrypted program code and a dongle. The “ideal model”, on the other hand, only allows Copy-and-Paste attacks. We can show that, given a program for which the VSA holds (and assuming the other three afore-mentioned assumptions hold as well), every adversary in the real model can be mapped to an equally effective adversary in the ideal model. More specifically, we can prove that for every real model adversary there exists an ideal model adversary that retrieves the same number of variants, while making the same number of dongle calls. The latter (same number of dongle calls) is important because an ideal model adversary that needs to make significantly more dongle calls than a given real model adversary is certainly not comparable to that real model adversary. Since ideal model adversaries are impractical for sufficiently large programs, this provides a meaningful security definition.

5 TOWARDS REALIZATION

As we have already mentioned, not every software can be protected effectively with our scheme. A software can be protected with our scheme only if it admits a partition into variants for which the VSA holds. Note that this property may only hold for some sufficiently *complex parts* of the program. Typically, such a complex part is exactly the critical part of the software that is needs to be protected. Moreover, the number of variants should be large enough so that the entire code

[1]See (Katz and Lindell, 2014) for a definition of IND-CPA security and EUF-CMA security.

cannot be traversed with a small set of input values.

We are currently exploring which code structures are suitable for protection. In particular, we are investigating how variants can be created in such a way that they fulfill the VSA and if such a variant creation process can be automatized. In this section, we briefly sketch some of our current research. We list some techniques that are useful for designing variants in such a way that the VSA holds and mention some of our findings on the efficiency of our scheme.

Techniques. In the following, we describe some techniques that are useful for creating variants that fulfill the VSA.

- *Local optimization:* Optimize the code for a particular input range, e.g., by deleting operations that are not necessary on a specific interval. For instance, one can delete additions with functions that are close to zero on a given interval.
- *Adding redundant operations:* Add redundant operations on a specific interval, e.g., multiplications with functions that are close to one on a given interval.
- *Approximation:* Create variants using approximation techniques (such as taylor approximation). Each approximation (variant) only gives local information about the original function block. If the original function is complex enough then an adversary cannot use this local information to deduce the entire function since the global behavior of this function may strongly differ from the local approximations. Note that it suffices if the original function is only complex on a particular critical part of its domain.
- *Exploiting branches:* Exploit the branched structure (e.g. if-then-else or switch statements) of the program code. If each branch contains code that is not easily implied by other branches, then one can define a variant for each branch.
- *Code Moving:* As mentioned earlier, the wrapper functions of each function block is moved into the dongle in order to hide the input ranges of variants. This technique makes it much harder for an adversary to find new variants. This is because, in general, knowledge of the input ranges of variants considerably facilitates finding new variants. In particular, without knowing the ranges an adversary cannot find out if he has retrieved all variants.

Implementation and Hackers Contest. In order to assess the practicality of our scheme, we have carried out various performance experiments together with an industrial partner specializing in the field of software protection. As it turned out, the dominating overhead is the latency of the communication between the computer on which the program is executed and the dongle. Variant decryption only plays a minor role. Our protection scheme does not use significantly more dongle calls than the (commercially available) protection schemes of our industry partner. This indicates that our scheme is practically viable. However, we still need to thoroughly benchmark our scheme in order to assess its exact efficiency.

We have carried out a global hacker contest with over 300 participants. Each participant received a small video game created by our industry partner and protected by our scheme. Every participant was given a dongle with a valid license that enabled running the protected program. No submitted solution contained functional program code. We believe that video games are among the classes of programs that can be effectively protected by our scheme due to their high complexity. This is because most modern games have a high number of "special cases" that are triggered by the behavior of the user. These special cases can be, e.g., a mission, a part of the balancing system, or an easter egg. In general, every complex code structure seems to be exploitable for variant creation.

6 OUTLOOK

The techniques for secure variant creation mentioned above are already applicable assuming the developer aids the variant creation process with his domain knowledge. Since the variant creation process should be as cheap and fast as possible, we are currently exploring ways to automatize secure variant creation.

REFERENCES

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. (2001). On the (im) possibility of obfuscating programs. In *Advances in cryptology – CRYPTO 2001*, volume 2139, pages 1–18. Springer.

Canetti, R. (2000). Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13(1):143–202.

Garg, S., Miles, E., Mukherjee, P., Sahai, A., Srinivasan, A., and Zhandry, M. (2016). Secure obfuscation in a weak multilinear map model. In *Theory of Cryptography Conference*, pages 241–268. Springer.

Goldreich, O. and Ostrovsky, R. (1996). Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473.

Joye, M. (2008). On white-box cryptography. In *Proceedings of the 1st International Conference Security of Information and Networks*, page 7.

Karroumi, M. (2010). Protecting white-box AES with dual ciphers. In *Information Security and Cryptology - ICISC 2010 - 13th International Conference, Seoul, Korea, December 1-3, 2010, Revised Selected Papers*, pages 278–291.

Katz, J. (2007). Universally composable multi-party computation using tamper-proof hardware. In *Advances in Cryptology - EUROCRYPT 2007*, pages 115–128. Springer.

Katz, J. and Lindell, Y. (2014). *Introduction to modern cryptography*. CRC press.

Lin, H. (2016). Indistinguishability obfuscation from constant-degree graded encoding schemes. In *Advances in Cryptology - EUROCRYPT 2016*, pages 28–57. Springer.

Lin, H. and Tessaro, S. (2017). Indistinguishability obfuscation from trilinear maps and block-wise local prgs. In *Advances in Cryptology - CRYPTO 2017*, pages 630–660. Springer.

Pinkas, B. and Reinman, T. (2010). Oblivious ram revisited. In *Advances in Cryptology–CRYPTO 2010*, pages 502–519. Springer.

Wee, H. (2005). On obfuscating point functions. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 523–532. ACM.