NEWS AND INSIGHTS FROM THE WORLD OF ID SECURITY

AUGUST 2023

# The VAULT

# QUANTUMania!

FEATURED ARTICLE

## Beware the Quantum Revolution!

### Quantum Computers Pose Grave Risk to Digital ID Security

Infineon Technologies

ALSO IN THIS ISSUE

# OBFUSCATION vs *ENCRYPTION:* Friends or *Foes*?

Dr. Carmen Kempka & Maurice Heumann, Wibu-Systems AG

So how do you protect information? A cryptologist would say: Nothing offers better protection than provably secure, well-studied methods based on sound assumptions. Kerckhoffs' principle states that security must never rely on the protection method itself being secret. A cryptosystem should be secure even if all information about it – except for a secret key – is known to the public. In contrast, software protection usually relies on obfuscating source code or executable programs as a defense mechanism against reverse engineering, an approach often dismissed as “security by obscurity”. But is it always that simple? And how bad can it be in practice?

We will examine software protection, and especially obfuscation, from both a cryptologist's and a software protector's point of view, thereby connecting the two worlds. This will put theoretical results about the effectiveness and limitations of obfuscation to the test of real-life experiences and attack vectors.

# Part I: The Cryptologist's view

## On the necessity of obfuscation

Given that well-established encryption methods like AES exist, why do we even bother with obfuscation in the first place?

Of course, executable programs can be protected by the accepted encryption methods. You can even have a more fine-grained protection by encrypting methods separately and decrypting only the parts you need at runtime. But the problem is, at some point, the CPU needs to get executable commands to actually fulfill the program's intended purpose. And points on an elliptic curve or elements of a cyclic algebraic group have the bad habit of not being executable on current hardware. So the curtain has to drop eventually, and a would-be attacker can seize on the plaintext executable statements for analysis. Cryptography cannot help here. At this point, the only line of defense to keep the adversary from trying reverse engineering is obfuscation.

## What is obfuscation?

But what is obfuscation? Barak et.al. provided a simple definition they call the "Virtual Black Box Obfuscation" (VBB), which characterizes ideal obfuscation with the three following rules:

- The obfuscated program should not be significantly larger than the original
- The obfuscated program should have the same functionality as the original
- The obfuscated program reveals no more information to the adversary than a black box would

Ideally, the obfuscated program should do the same as the original would, without too much overhead impacting performance or the program size, while all the adversary learns from the obfuscated program is input-output behavior, which they could learn anyway by simply executing the program.

## Obfuscation is impossible!

Unfortunately, in the same paper, Barak et.al. proved that it is impossible to design an obfuscator that meets this definition. But is this final proof that obfuscation is, in fact, bound to be no more than the frowned-upon "security by obscurity"?

The intention was never to disprove the purpose of obfuscation, but to explore the limits and possibilities of this, until then, oft-ignored complementary concept to cryptography.

Let us take a closer look at the actual paper's findings. What it really says is that there is no general obfuscator that can obfuscate every existing program in the VBB sense. This does not mean that "no program can be obfuscated". It rather means that "there is one program that cannot be obfuscated". Or, as stated in the paper: "As is usually the case with impossibility results and lower bounds, we show that obfuscators (in the "virtual black box" sense) do not exist by supplying a somewhat contrived counterexample...".

In fact, there are functions (so-called "point functions") that are obfuscatable in the VBB sense, for example a password check.

But even for other functions, all hope is not lost. An alternative definition called "indistinguishability obfuscation" has been proposed to overcome the impossibility of VBB, and it has been proved achievable with several candidates for obfuscators already constructed. While this was an important step towards closing the gap between theory and practice, these constructions are still quite far from being practical. Running an AES encryption with one of these constructions would, for example, take not less than 272 years and consume several petabytes of storage.

## What can we learn from cryptography?

Given that we have all the experience from designing encryption algorithms at hand, how can we use it to close the gap?

Encryption algorithms are usually based on hard mathematical problems, like the problem of factoring large numbers used for RSA or the discrete logarithm problem used in elliptic curve cryptography. To use a similar approach for obfuscation, we need to overcome several obstacles. The result of obfuscation is, for example, usually supposed to be executable on a CPU as it is, while ciphertext has to be decrypted before reading. So we need to find a hard problem for transforming executable code into obfuscated, but executable code that keeps the same functionality.

The good news is that there are actually NP-hard problems that can be and are used for obfuscation. One of these is the SAT problem. In the obfuscation world, this usually comes as opaque predicates or the problem of dead code elimination: the adversary is deceived by a lot of if statements, and needs to decide which of these always evaluate to "false" and are thus never executed. Another example is using different pointers, called aliases, to access the same value.

Unfortunately, these NP problems are difficult only in the worst (or for us, best) case. A randomly chosen instance of the SAT problem, for example, can usually be efficiently solved with a logic solver.

For encryption algorithms like RSA, we have learned how to choose good key pairs, i.e., instances of the underlying problem which are actually difficult for our adversary to solve. For obfuscation, this is still an open problem, and many of the commonly used obfuscation techniques can, if considered one by one, eventually be cracked by attackers. In addition, understanding the complete program might not even be the goal of our adversary, as they might just want to eliminate a license check.

## How far are we from what we actually need?

Encryption algorithms are usually designed in a way that the adversary would need millions of years or more to solve the underlying problem, and that there is a significantly low probability of correctly guessing a secret key. In software protection, depending on the use case, we might just want to keep our adversary from cracking anything until a new version of the software is published, or until the bulk of our prospective income has been made with the protected software.

But even if currently known obfuscation techniques don't (provably) achieve the same level of protection as encryption methods, and even if single protection techniques can be broken, the attacker is not necessarily able to crack our software. Multiple obfuscation and integrity protection techniques can be used to protect each other, achieving a very strong level of protection as long as, for each attack technique, there is a protection technique which prevents that attack. This opens the same kind of cat-and-mouse game between attackers and protection that we already know from cryptography, and that we invite you to experience in the rest of this article.

**DR. CARMEN KEMPKA** *studied computer science at the University of Karlsruhe (TH) with a focus on cryptography and quantum computing. After completing her PhD in 2014, she joined the Secure Platform Laboratories of NTT in Japan for two years of postdoctoral research in cryptography. She moved to WIBU-SYSTEMS AG in 2016, where she is now responsible for R&D projects and supports her colleagues with all questions about cryptography and security.*

# Part II: The Protector's view

**MAURICE HEUMANN** *studied computer science at the Baden-Wuerttemberg State University. He began programming at the age of nine and started working with reverse engineering at 14. Since 2019, he has been working as a protection engineer at WIBU-SYSTEMS AG, where he develops and improves software protection solutions. In his spare time, he reports software vulnerabilities to renowned game manufacturers.*

## Where does the attacker start?

In contrast to the theory, real-life attackers often begin their analysis by examining programs in a disassembler. To counteract this, software can be packed, using compression or encryption, and then unpacked during runtime. However, as previously mentioned, it is important to note that the application code will eventually need to be decrypted for execution. This creates a window of opportunity for attackers to inspect and dump the memory contents after decryption, allowing them to analyze the code and carry out runtime patches, which is called binary hooking. Without employing obfuscation techniques, it is not possible to prevent attackers from analyzing program semantics.

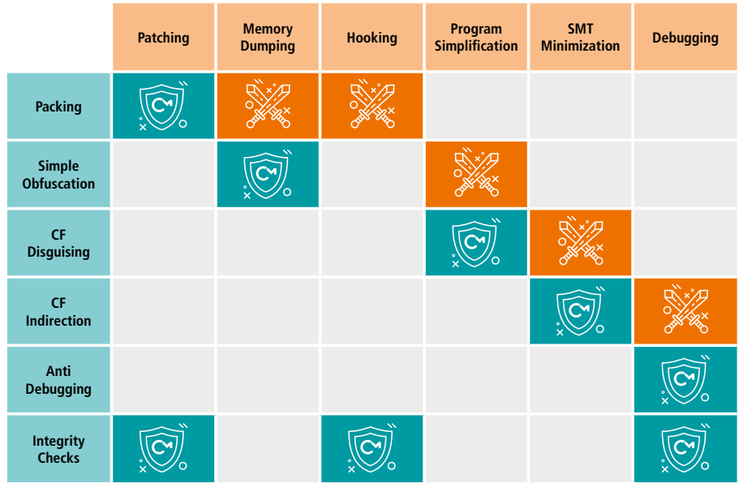
## What does obfuscation look like in practice?

For greater security, attackers should be prevented from actually comprehending the underlying code. One should not rely merely on obstructing their analysis. Simple obfuscation techniques can be employed to help us some way towards this objective. These techniques involve injecting redundant instructions or substituting instruction sequences with more complex forms. By employing such obfuscation methods, it gets harder for attackers to understand the code, providing an additional layer of protection for the program.

**Program simplification** While simple obfuscation techniques provide some initial barriers, they are rarely sufficient to impede determined attackers from comprehending the program logic in the long term. Attackers can utilize program simplification methods to streamline obfuscated code. This involves lifting the obfuscated code into an abstract language known as intermediate representation (IR). The IR can be subjected to various optimization techniques commonly employed in compiler frameworks, resulting in a simplified form of the program. This optimized representation can be either translated back to assembly code or visualized through decompilation, making it more accessible for attackers to analyze and understand it.

**Control flow disguising** To address the limitations of program simplification and the performance impact caused by inserting redundant code, control flow disguising techniques offer a viable solution. Instead of substituting instruction sequences with complex forms, dead code is introduced into the program. This dead code can appear arbitrary or resemble the original code. The connection between the dead code and the original program is established through the use of opaque predicates. These opaque predicates take the form of conditional statements whose evaluations consistently yield a fixed result (either true or false). With that, the dead code is never executed, and the performance impact is minimized, with the exception of the opaque predicates themselves.

**SMT-assisted minimization** To overcome the challenge posed by opaque predicates, attackers can employ SMT-assisted minimization techniques. SMT, short for Satisfiability Modulo Theories, generalizes the boolean satisfiability problem (SAT). By utilizing symbolic execution, attackers can transform





## What about dynamic analysis?

While the mentioned obfuscation techniques effectively hinder static analysis, dynamic analysis remains a viable avenue for attackers. For instance, reconstructing the program's control flow by debugging it is still possible. As the code must maintain semantic equivalence and all relevant code blocks must eventually be executed, attackers can utilize a debugger to step through the program's execution and reconstruct the control flow. By observing the program's behavior during runtime, attackers can gain insights into its control flow and understand the underlying logic.

To counteract the presence of debuggers at runtime, various anti-debugging techniques have been developed. However, many of these techniques are widely known and considered ineffective, as tools exist that can automatically bypass or disable such anti-debugging measures.

A more robust approach involves the use of integrity checks to protect against code manipulations. By computing a checksum of the code at runtime, programs can verify that their code has not been altered. In the event of a mismatch, appropriate actions can be taken to prevent further execution.

Integrity checks are highly effective in combating patching and hooking techniques, as any modifications to the code are automatically detected. Furthermore, these checks also provide protection against debugging attempts. Debuggers typically insert breakpoint instructions to pause program execution. These instructions are detected by the integrity checks, thereby preventing successful debugging.

## How can obfuscation be implemented?

Finding the right defense techniques against attackers is an important task. However, without the ability to incorporate these techniques into software programs, their potential remains dormant.

Leveraging compiler frameworks offers a practical solution to that. Compiler frameworks, such as LLVM, offer the ability to parse, optimize, and lower code for specific architectures. By intervening in the optimization, additional obfuscation and protection techniques can be seamlessly inserted into the code.

Wibu-Systems offers AxProtector CTP, a powerful product that leverages the LLVM compiler framework to achieve efficient obfuscation goals. With AxProtector CTP, a wide range of defense techniques, including those mentioned earlier, can be seamlessly integrated to enhance the security of various applications. Additionally, it provides flexible licensing features supported by trusted cryptographic algorithms. The synergy between licensing, encryption, and obfuscation ensures optimal protection for applications.

Thanks to the versatility of LLVM, AxProtector CTP supports multiple operating systems, architectures, platforms, and programming languages. Compared to other established protection techniques, AxProtector CTP offers non-invasive application security. It adheres to code integrity requirements, such as those enforced by Apple, on Apple silicon machines. This ensures that applications remain secure while maintaining compliance with platform-specific guidelines, even without the need for runtime code modification. ⊠

WIBU SYSTEMS

# CodeMeter – A Symphony of Software Monetization Tools

- Compose your original code
- Orchestrate your license strategy
- Fine tune your IP protection
- Distribute your work of art

Sounds easy, right?
And it is with CodeMeter

CodeMeter Cloud
CodeMeter Certificate Vault
CodeMeter Embedded
CodeMeter Dongle
CodeMeter LC
CodeMeter

Start now and request your CodeMeter SDK
wibu.com/sdk

+49 721 931720
sales@wibu.com
www.wibu.com

SECURITY
LICENSING
PERFECTION IN PROTECTION