SCTE·ISBE CABLE-TEC EXPO 2019 NEW ORLEANS, LA SEPT. 30-OCT. 3

2019 Fall Technical Forum SCTE·ISBE • NCTA • CABLELABS

Dynamically Addressing the Gap of Software Application Protection without Hardware Security

A Technical Paper prepared for SCTE•ISBE by

Rafie Shamsaasef
Director of Software Engineering
CommScope
6450 Sequence Dr.
San Diego, CA 92121
(858) 404-2205
rafie.shamsaasef@commscope.com

Aaron Anderson
Senior Security Architect
CommScope
117 St. Georges Bay Rd., Parnell
Auckland, New Zealand

+64 935 803 75
aaron.anderson@commscope.com

Table of Contents

List of Figures

Introduction

Hardware security may not always be available or feasible for applications running on platforms ranging from Original Equipment Manufacturer (OEM) devices to public cloud servers. To resolve this dilemma, a comprehensive software security solution is required that is easily applied and readily utilized by developers. This solution would address the security gap created by a growing demand for quickly deployable and securely protected applications. The authors will discuss newly developed advanced security technologies to provide practical protection against a wide range of attacks. These technologies deliver another layer of security to protect sensitive data and credentials.

With the exponential growth of video distribution to millions of subscribers, the processing and secure delivery of video is now more than ever essential to programmers, developers, and operators.

Utilizing a combination of innovative solutions such as white-box cryptography, software obfuscation and code signing, this flexible solution balances protection and performance while allowing customers to design, code and build to suit their needs. This is especially true in an end-to-end media content distribution system where attacks are often aimed at defeating conditional access or finding ways of exploiting services that are easier than circumventing cryptographic protection.

The design secures cryptographic algorithm implementations against intrusions such as secret-key, code-lifting and side-channel attacks and allows the implementation of standard ciphers such as RSA, AES and ECC so that no intermediate key or data is exposed during cryptographic operations. The solution recognizes the threats of reverse engineering, debugger attachment and tampering attacks, and creates tools to further create a layer of security.

The authors provide insight into this comprehensive security solution and underlying technologies that protect software applications.

Content

1. Security Basics

1.1. Holistic Approach to Application Security

A holistic approach to security seeks a comprehensive, systematic, and interconnected view of safeguards and protections to the entire application software. For security to be considered holistic, many requirements must be met to make separate areas of security compatible and interoperable. The integration of different levels and types of security enables a more comprehensive understanding of vulnerabilities and more comprehensive protection against a variety of threats [1]. This can only be done by breaking down various software components to independently analyze and then look at the interactions among these modules to identify potential threats.

The holistic view starts from the top-most interfaces of the application interactions and bubbles down to the lowest atomic security operations. Protections and safeguards applied to software modules in different layers can differ in coverage range and severity of their security. Not all software modules are entitled for highest level of protection. That's where the comprehensive attack surface analyses play an important role in deciding what is the "secure enough" protection for each module.

There is no silver bullet in solving application protection problems. Cryptography or special security features do not magically solve all security problems rather the security is emergent property of entire system and should be viewed just like quality. Therefore, security aspects should be integral part of the design, right from the start as highlighted by Erik Poll, Digital Security group of Radboud University.

As you design your application, you need to be aware of potential vulnerabilities of each software module and layer composing your application. Today's software is built on the top of mixture of operating systems, platforms, libraries, programming languages, 3rd party unitalities, tools, IDEs and compilers that make it hard to trace security holes. The holistic approach to application security requires designers and programmers to examine and understand this complexity knowing that there is always a trade-off decision to be made as security is never going to be 100%. Nevertheless, it is possible to achieve certain level of satisfactory protection by deploying a combination of techniques as discussed in more dept in this paper.

1.2. Software vs Hardware Security

There is always a trade-off between software and hardware security. This is more apparent in today's dynamic software markets. The ecosystem contains various devices and platforms where not all software vendors have access to hardware resources, and it may not be feasible or cost effective to use hardware security. Even if the decision is made to use hardware security, then the questions arise such as, "How much of it? Where do you draw the line? Should you use hardware root of trust? Should you use just hardware acceleration for crypto operations or implement trusted application (TA)?" With many other questions and there is no one solution fit-all answer.

A comprehensive analysis is required to identify the benefits and the trade-off of choosing software vs hardware security. The answers depend on the sensitivity of the data and operations on those secret data within the application. Whether to opt for software-based or hardware-based solutions should be one of the early decisions designer and developers are faced with, and it's not an easy choice. Although both technologies combat unauthorized access to data, they do have different features and must be evaluated carefully before implementation. "Software is easier because it is more flexible, and hardware is faster when that is needed" [2].

Hardware security always relies on the strength and capabilities of the targeted platform. This comes with strings attached for implementation and usage of platform specific instructions and interfaces that are not typically portable to other hardware. There are security processors with dedicated memory regions to handle sensitive operations for data and to hide them from the main processors and memory area where other applications execute. This trusted execution environment (TEE) provides a great deal of security and protection for your application, however it comes with certain restrictions. For instance, the code will not be portable to other hardware unless the same chipset category by the same vendor is used. The trusted application (TA) which is running in TEE is not easy to develop and debug. Therefore, it takes special skills and expertise than regular software engineer to do the job. Careful analyses and evolutions must be done before embarking on the hardware security path since the swift change of direction could be costly. There is wide range of alternative for hardware-based and software-based implementation of security as shown here.

Figure 1 - Hardware vs Software Security Spectrum

Software is viewed by many experts as a source of security problems and the weakest link of the security chain. However, the proper implementation of it could provide enough protection for certain applications.

When is this feasible? When software solutions make use of shared memory space, are running on top of an operating system, and are more fluid in terms of ease of modification [3]. Although the software solution to security doesn't possess the same caliber strength of hardware security, it can provide enough assurances to offer a certainly sustainable and viable alternative.

There are also mandates by government, industry associations or a company to be adherent to certain robustness rules. Even though the choices are very limited under those circumstances, the concept of the holistic approach to the security can still be applicable

With the advent of white-box cryptography and software obfuscation, the risk of exposing keys and sensitive data during cryptographic operations are minimized and mitigated by the strength of these technologies. The intend of this paper is to explore what makes software security a viable alternative to hardware security for application development.

2. White-Box Cryptography

2.1. What is White-Box?

The goal of white-box cryptography is to protect cryptographic keys against attackers who typically have full control and visibility of software running on untrusted devices. Compared with pure hardware alternatives, white-box protection can be cost-effectively installed on any device and can be readily updated [4].

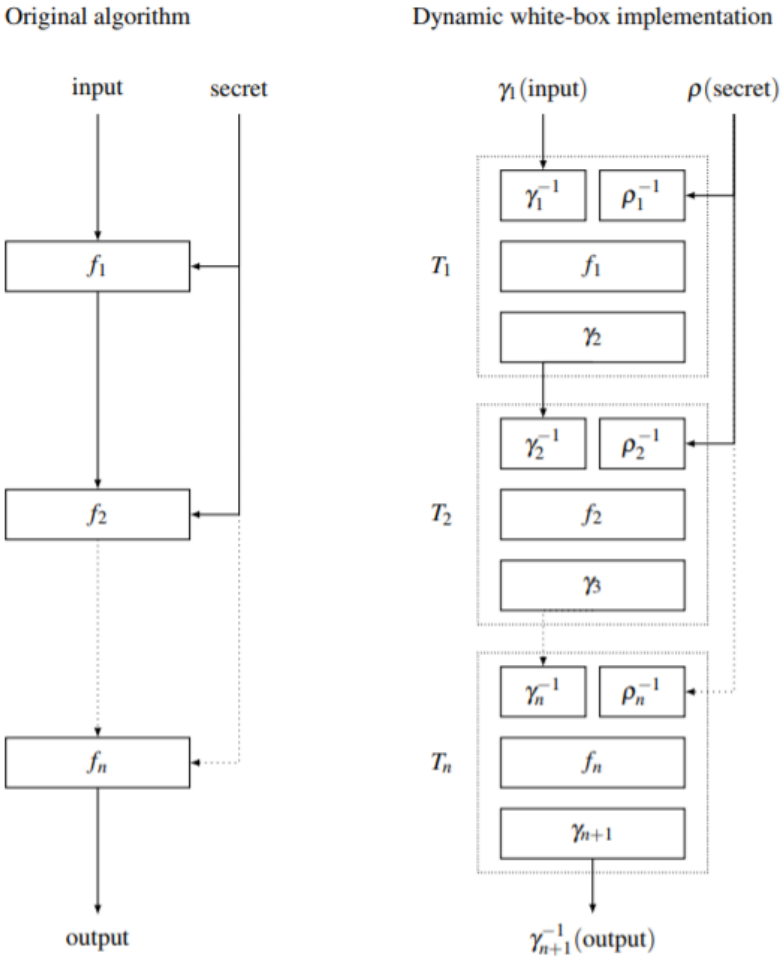

Figure 2 - Table-Based White-Box Transformations

In Figure 2, table-based white-box transformations compose random bijections with an application's functions. These compositions are emitted as lookup tables so as to conceal the underlying secrets and other state values.

2.2. Core Criteria for Application of White-Box Crypto

The intuitive security notions of table-based white-box implementations have been formalized into concrete security notions with the introduction of a *white-box compiler* that turns symmetric encryption schemes into randomized white-box programs. This has enabled the capture of additional formal security properties, such as one-wayness, incompressibility, and traceability for white-box programs [5] [6].

White-box compilers are also used to broaden the scope of white-box protection to include compatible non-cryptographic application code (such as codecs) in the white-box scheme [7].

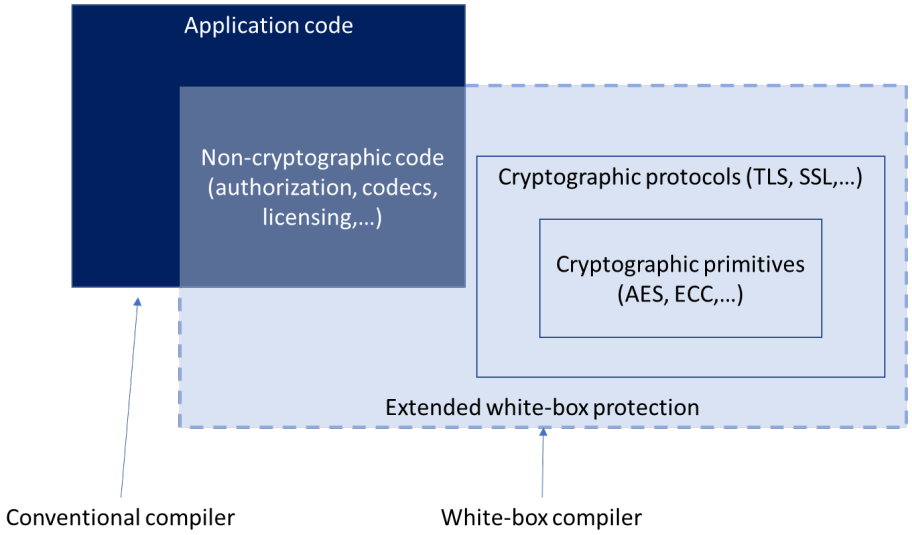

Figure 3 - White-Box Compiler

In Figure 3, a white-box compiler is used to protect sensitive cryptographic and non-cryptographic components in an application.

It must be noted that a white-box compiler is not a general-purpose compiler, nor is it an obfuscation method in the usual sense, thus the candidate algorithms must have mathematical properties that are conducive to white-box encoding.

2.3. White-Box Security

The initial constructions and later variations of Chow, et al.'s white-box implementations have all been broken theoretically [8] [9] [10] [11]. The *WhibOx* competition, launched by ECRYPT CSA as the capture the flag challenge of CHES 2017, ended with all 94 submitted white-box implementations broken by practical attacks. Only 13 implementations survived for more than one day. These theoretical and practical results demonstrate that attackers prevail in the present-day cat-and-mouse game of white-box security [12].

Despite these results, it must be noted that conventional black-box implementations of even the theoretically strongest possible algorithms fall to essentially zero security on hostile platforms, as their secret keys are directly observable by an attacker [13].

Message security	Strong	Strong
Implementation security	Zero	Weak

Figure 4 - Security Properties of Black-Box Versus White-Box Security

2.4. Code Lifting Attacks

Besides known cryptanalytic weaknesses, defining white-box security as the impossibility to extract the secret key also has some drawbacks. Namely, it leaves the door open to code lifting attacks, where an attacker simply extracts the white-box implementation as a whole and achieves the same functionality as if he or she had extracted the secret key [14].

White-box node-locking techniques can be employed to restrict the operation of a white-box to a device by encoding every white-box operation to that device's signature. When used in conjunction with external encodings, which push the boundaries of the white-box into surrounding applications, this method is shown to be effective in mitigating code-lifting attacks [15].

Further research is focusing on new security notions, such as *space hardness*, which give a quantifiable measures of security against key extraction, table decomposition, and code-lifting attacks [16] [17].

2.5. White-Box Chaining

The use of external input and output encodings in white-box implementations facilitates the notion of white-box chaining, where the output from one white-box implementation can be used as input into another without an intermediate cleartext step. This allows architectural and implementation flexibility, of white-box components that may be housed within different applications, on different physical devices, or in cloud-based environments [15] [18] [19].

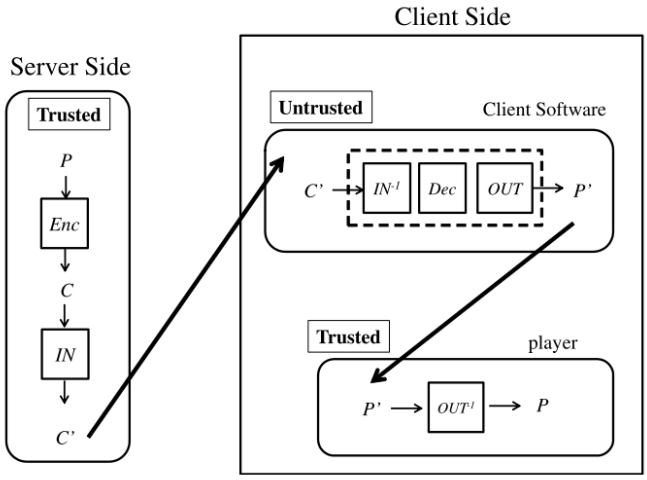

Figure 5 - External Encodings

Figure 5 illustrates external encodings used to chain implementations in a DRM setting [17]. Such encodings also allow white-box implementations to be securely bound to applications and hardware security devices [15] [18].

2.6. Moving Toward Strong White-Box Cryptography

Strong white-box models seek to better emulate trusted hardware execution environments in software by finding efficient *virtual black box* or *indistinguishability obfuscation* constructions, thus circumventing the cryptanalytic weaknesses of present-day white-box implementations. Models with slightly weaker security properties permit adversaries with access to white-box implementations to encrypt (at least via code lifting) yet remain unable to decrypt [14].

Notable progress has been made with a *fully homomorphic white-box (FHWI)* construction that offers efficient strong white-box implementations of specific applications without utilizing the lookup-table structure of Chow et. al. These implementations are shown to achieve semantic security [5].

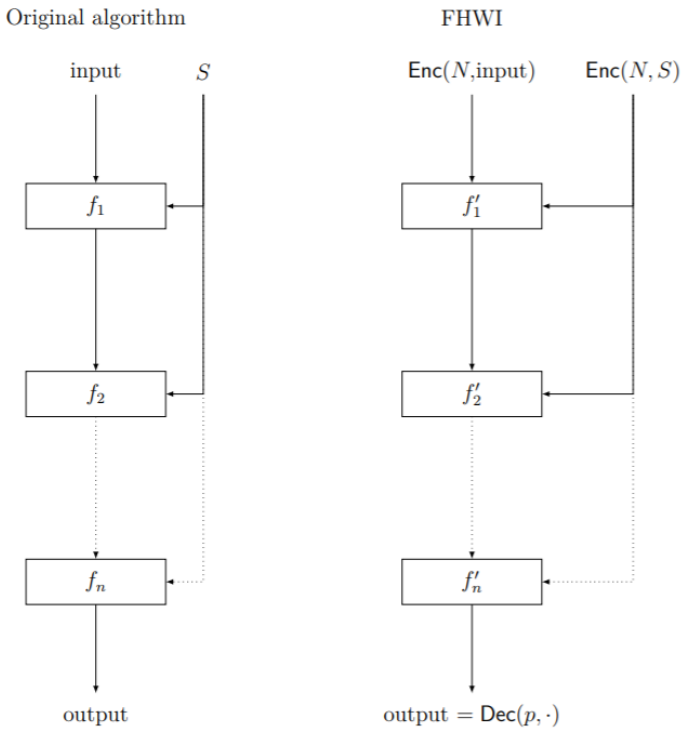

Figure 6 - FHWI Transformations

Figure 6 illustrates FHWI transformations of functions into strong white-box functions in a homomorphic white-box domain with semantic security (where it is infeasible for an adversary to extract any information about the secrets and values under computation, even with full visibility and control of the execution environment.)

3. Software Obfuscation

3.1. What is Obfuscation?

Software obfuscation aims to make programs "unintelligible", while preserving their functionality, in order to prevent reverse-engineering of their intellectual property and to help mitigate tampering attacks. This has been an active area of development for decades, with a wide range of solutions currently on offer. In many cases, these obfuscation systems are proprietary or lack a rigorous theoretical underpinning, which makes their security properties difficult or impossible to validate.

The theoretical study of program obfuscation began with the introduction of a formal definition of *virtual-black-box obfuscation* by Barak, et al. [20] in 2001, which captures the requirement that such an obfuscator can securely hide information about a program. Barak, et al. went on to show that it is impossible to achieve general-purpose virtual black-box obfuscation.

3.2. Control Flow Obfuscation

Control-flow obfuscation is a theoretically tractable approach to obfuscation that transforms the structure of a program's control-flow graph in such a way that it cannot be easily reconstructed by static analysis; thus hindering the comprehension of the program by an adversary. The most prevalent form of control-

flow obfuscation is *control-flow flattening*, where a program is reduced to a garbled collection of basic blocks with an execution sequence based on a state variable that is dynamically computed at runtime. This has been shown to be a PSPACE-complete problem [21].

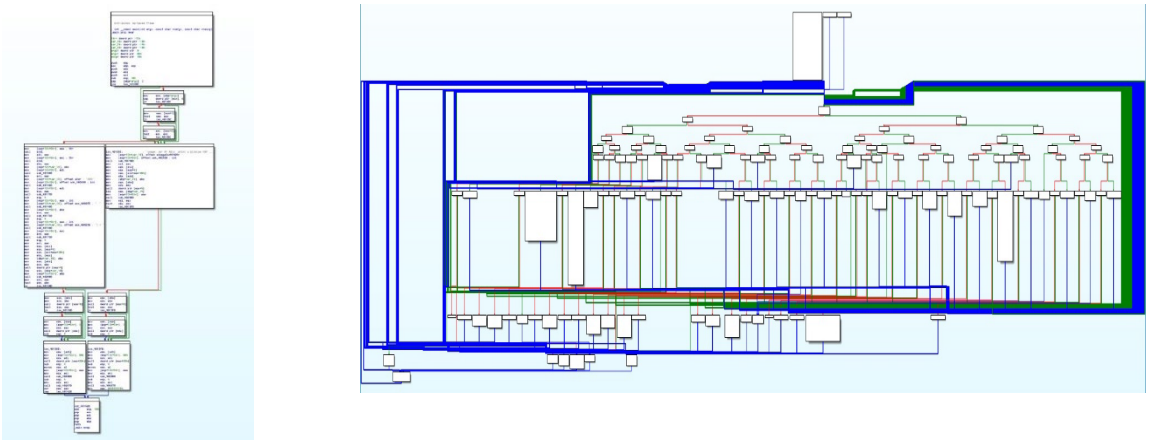

Figure 7 - Control-Flow-Flattening Example

Further theoretical advances in this area utilize a *virtual machine interpreter* to add an abstraction layer between the labels used to identify basic blocks by control-flow flattening and the underlying memory addresses of those blocks. This has been shown to be a NP-complete problem [22].

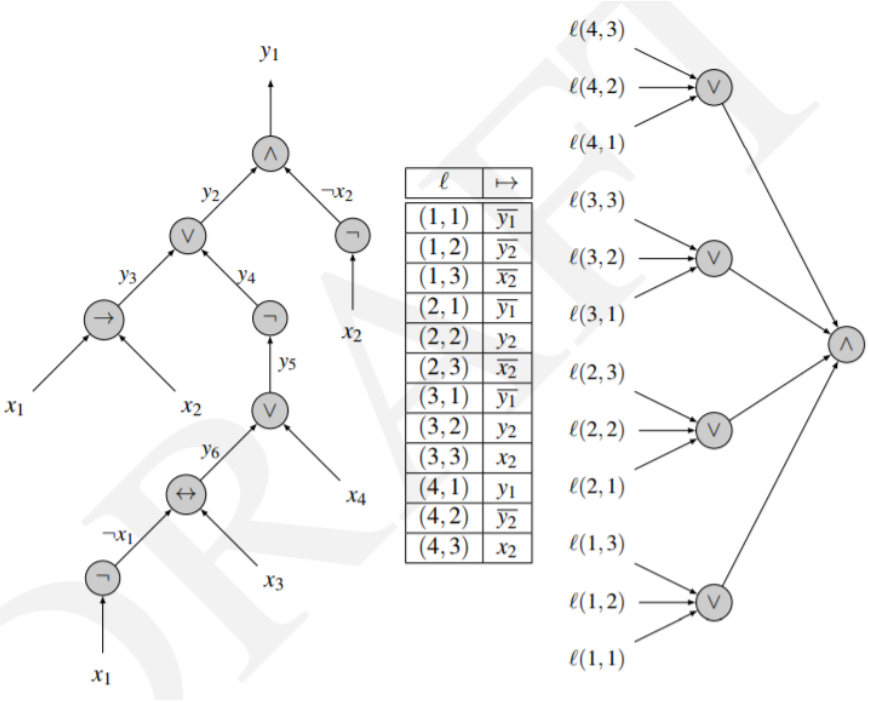

Figure 8 - 3-CNF-SAT Reduction of a Virtual Machine Interpreter

3.3. Data-Flow Obfuscation

The goal of data-flow obfuscation is to achieve maximal unintelligibility of the dataflow of an obfuscated program. Advances in this area utilize algebraic structures that facilitate *structure-preserving randomization* of the data-oriented operations of a program. In this model, Boolean and arithmetic operations are encoded as randomized sequences of matrix operations without altering functionality [23].

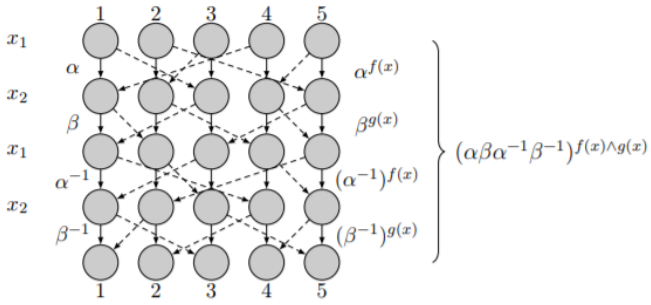

Figure 9 - Branch Encoded Function Computing an AND Gate

Figure 9 Illustrates a branch encoded function computing an AND gate. Such randomized encodings are used to create diversity and maximize unintelligibility. By randomly encoding and merging data-flow operations, this technique produces a "confusion" effect. When used iteratively together with control-flow obfuscation, which can be likened to a "diffusion" effect, produces synergistic obfuscation results that are highly complex (entropic), in a similar manner to confusion and diffusion operations in a *substitution-permutation network* in classical cryptography.

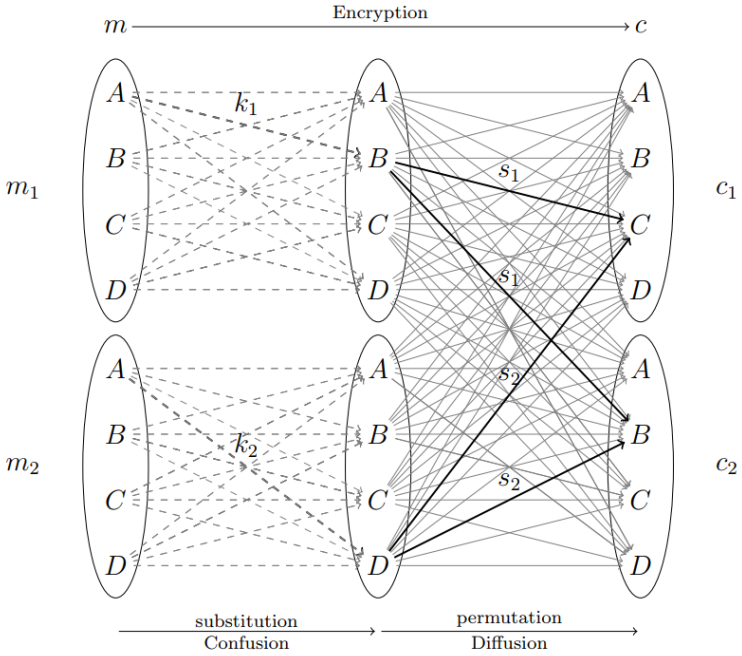

Figure 10 - Confusion & Diffusion

Confusion is introduced by uniquely substituting inputs. Diffusion is then introduced with a permutation that combines the results of each substitution. These steps are repeated to achieve maximal complexity in the output.

3.4. Strong Obfuscation Models

In order to avoid their impossibility result, Barak, et al. defined a variant obfuscation model called *indistinguishability obfuscation* (iO). Finding a practical implementation of iO as well as finding ways to circumvent the general impossibility result are central topics in cryptography today [24].

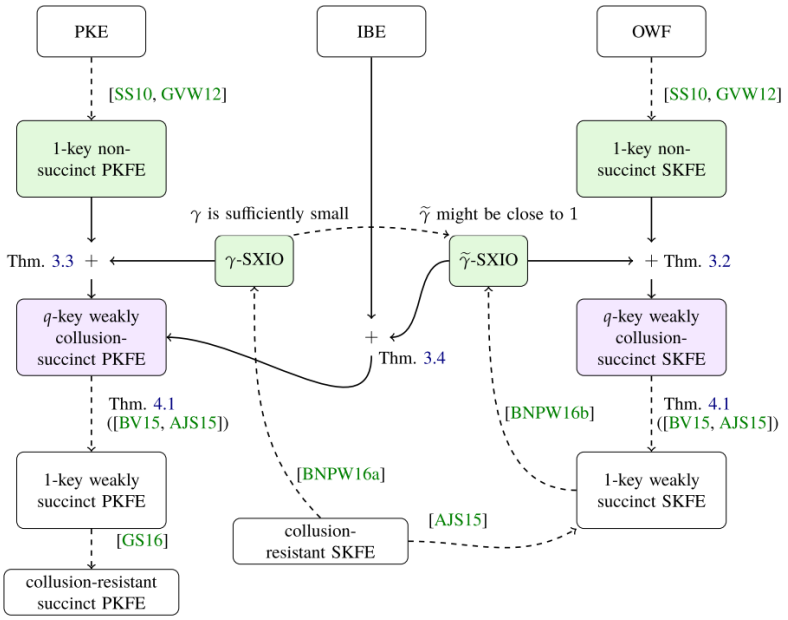

Figure 11 – Research into efficient iO

Due to its importance, achieving efficient iO is a central focus of present-day cryptography research [24].

4. Other Software Protection Techniques

There are a range of synergistic techniques that compliment present day white-box cryptography and obfuscation models to harden these approaches against common attacks. These methods may be unnecessary under stronger white-box and obfuscation constructions that are presently under development.

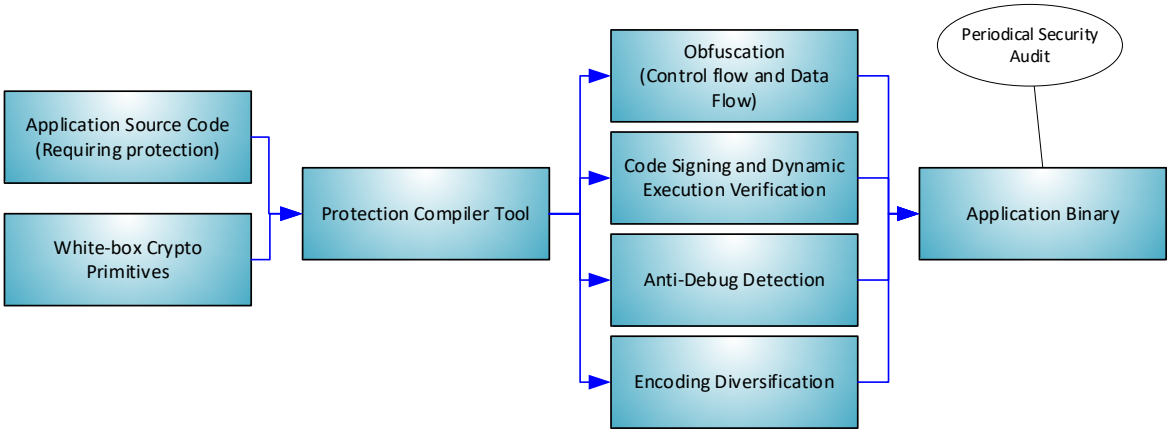

Figure 12 - Protection Process

4.1. Anti-Debug Protection

Software that employs anti-debugging techniques can determine if it's being debugged by identifying artifacts of the debugging process [25]. Detection and response code is injected into an application to

counter debugging attempts. Control and data-flow obfuscation can then be applied to further increase the stealthiness and diversity of the detection code.

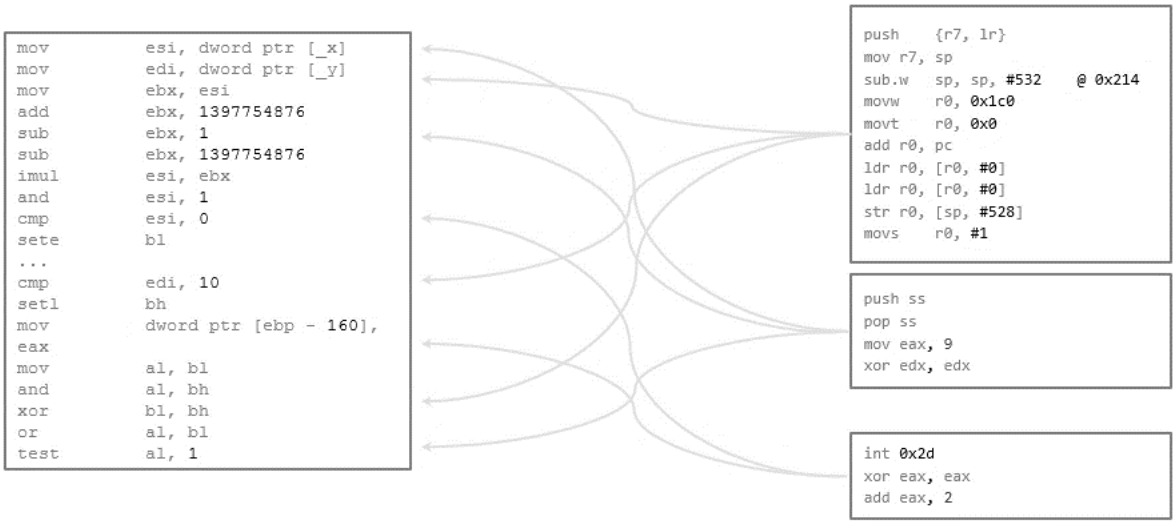

Figure 13 - Anti-Debug Detection Code

Figure 13 provides a visual example of randomly selected anti-debug detection code (right) being injected into random points of a program (left).

4.2. Code Signing

Code signing methods in commercial use are targeted at preventing *static tampering attacks,* which involve unauthorized modifications to a program's binary code prior to execution [26] [27].

4.3. Dynamic Binary Image Verification

Static code signing methods do not detect modifications made to executable code at runtime, such as with *buffer overrun attacks*, which are some of the most prevalent tampering attacks in today's landscape [28]. Advances have been made in *dynamic integrity protection* with the goal of operating in a stealthy manner, minimizing false positives, not overly impacting performance and maintaining full compatibility with static code signing across a wide number of platforms [29].

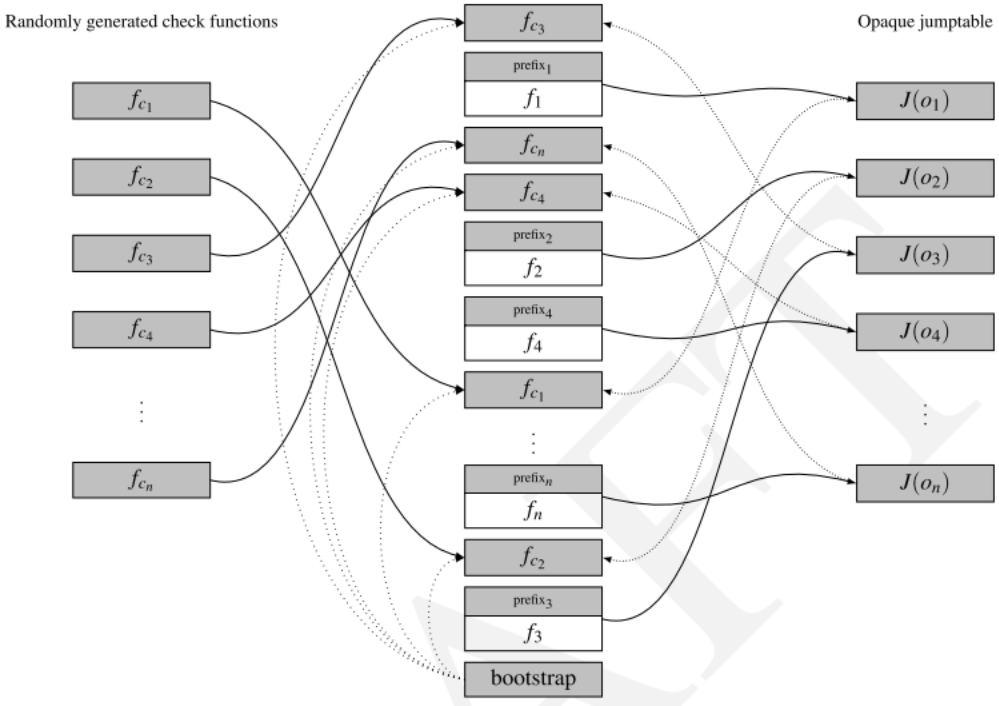

Figure 14 – Implementation of Dynamic Integrity Verification

Figure 14 is an illustration of a stealthy implementation of dynamic integrity verification with randomized check functions (left), randomized check points (middle), and opaque jump table to conceal their relationship (right).

4.4. Diversification Techniques

Software diversity refers to generating unique software implementations that deliver the same functionality. This approach introduces uncertainty against a wide range of attacks. This uncertainty can be expressed probabilistically in a manner similar to cryptography, where the success rate of any given attack can be expressed as a function of the entropy of the implementation. Researchers have proposed multiple approaches to software diversity that vary with respect to threat models, security, performance, and practicality [30].

Software diversity is not just a simple layer on top of existing security, it is fundamental to every part of the security model [30]. Examples of compilation diversification this can be seen in Figure 13 and Figure 14, where randomly generated detection functions are injected at random points in the code to ensure

maximal uncertainty of the type and location of those code blocks, hence maximizing the diversity of their implementation.

4.4.1. White-Box Diversity

White-box implementations (produced by a white-box compiler) are essentially a network of randomized lookup-tables [19]. Hence, such implementations are maximally diverse.

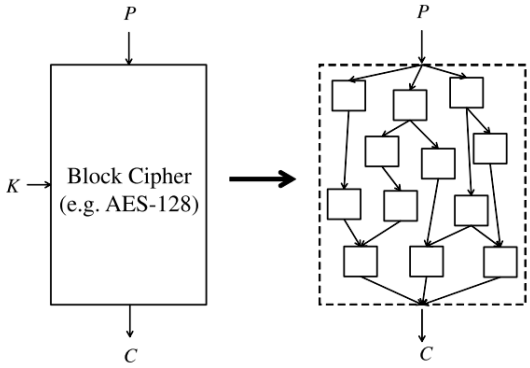

Figure 15 – Diversity in a Table-Based White-Box Implementation

Table-based white-box implementations are a randomized network of randomized table look-ups [17].

4.4.2. Diversity from Obfuscation

Control and data-flow obfuscation further increases diversity and as such can be applied iteratively and in conjunction with other techniques, such as anti-debug and dynamic integrity protection methods.

- Control-flow obfuscation spreads out (flattens) a program's control pathways in a random manner, thus introducing uncertainty of the location of code elements.
- Data-flow obfuscation randomizes and merges arithmetic and Boolean instructions, thus introducing uncertainty of function.

The security of these notions can be formalized in terms of the entropy of a software implementation [30].

4.4.3. Diversity as Side-Channel Protection

One of the key requirements to build a side-channel attack is the ability to accurately replicate the victim environment. However, software diversity breaks exactly this assumption. Attackers no longer have easy access to an exact copy of the target program [30].

4.5. Periodical Audit

Another key element of a robust security model is the ability to forensically examine usage, exceptions, and critical failures across the entire spectrum of protection. Again, these metrics must be embedded in each security module to generate data that can be acted on appropriately.

5. Application Threats and Security Analysis

5.1. Cloud Application Threats and Related Top 10 OWASP Threats

The Open Web Application Security Project (OWASP) has been around for various types of software applications for years. Recently, the Cloud Security project was added to help people secure their products and services running in the cloud. This project provides a set of easy to use threat and control Behavior Driven Development (BDD) stories that pool together the expertise and experience of the development, operations and security communities.

With growing cloud computing elastic platforms that are feature-rich and highly scalable, companies are now able to deliver products and services with a velocity and agility that has never been seen before. This could sometimes result in circumventing security thus the rise new attack vectors [31]. Considering OWASP Cloud Security recommendations and best practices are great starting points for security architects to analyze their applications and come up with attack surface.

However, to make this more effective, the process must be iterative with tangible feedbacks. As a new security feature is designed and implemented, it's possible that the existing attack surface is expanded and must be analyzed again. Then a new set of penetration test should be considered and executed to have collect feedback. These test results should then be fed back to the designer/developer to analyze the impact and the risk of new features. This iterative process should be part of ongoing practice of application development as we explore more later in this paper. It allows examining the impact of each added security feature until a level of satisfaction is achieved.

5.2. Systematic Approach to Security Analyses Process

There are three aspects to security analyses: *Threat, Assets, and Vulnerability*. Threat refers to various ways to potentially break and exploit an application. Assets refer to secret data, metadata, and sensitive procedures used by application to perform its job. Assets can come in the form of static data or dynamic run-time data temporarily stored in the memory. Vulnerability is an existence of a weakness, design, or implementation error that can lead to an unexpected and undesirable event compromising the security of a system. Vulnerability is essentially a hole in the application that causes it to provide a gateway to break it. The *security risk* is the intersection of these three aspects where a threat is apparent on an asset thru vulnerability of the application in the module where the asset is handled.

The impact and cost of not engaging with security analyses systematically are huge. To name a few:

- Technical and business impacts
- Credential data lost; metadata lost
- Personal data exposed; app data exposed; enterprise data exposed, etc.
- Measurable and indeterminate damages that comes with system repair cost; outage costs

Understanding these impacts helps application designers/developers better analyze what kind of protection is needed in every area of application. There is no perfect answer as applying security and protection comes with its own set of costs and complexity such as performance impact, overhead of Enc/Dec & network authentication, and complex troubleshooting especially for TEE environment.

5.3. Incremental Protection Value Analysis

The incremental protection value analysis is an analytical method in the systematic approach to security. Its purpose is to identify the acceptance level of protection techniques as applied to certain data and procedures within the software application. *There is no technical limit to the level of security that can be*

applied to assets or modules. The analysis includes wide range of techniques from TEE hardware-based security to all software-based implementation as discussed before. The incremental protection value analysis attempts to assign a numeric value to a combination of protection technique and targeted data/procedure. It is a method of measuring a protection's strength at any given point statistically or dynamically in run-time. It compares the intended amount of protection effect with what has been deployed with the application. It is achieved by security testing and proactively applying the known and common attacks to your application and measure its resilience. This allows application designer/developer to evaluate level of protection while having enough guards against associated attacks. It is the striking balance between protection and threat.

Here are steps to perform gained value analysis:

1. Determine the strength of each protection technique within your domain.
2. Sort and rank these techniques based on known risks and attacks.
3. Associate a security strength value to each protection technique. This is a ranking value from the lowest level (no-security) to the highest level (in your application domain).
4. Identify assets and procedures needing protection and associate a sensitivity/secrecy value to them. The higher the number, the more secrecy the data or more sensitive your procedures.
5. Start assigning an acceptable range of protection strength to your data/procedures with a comfortable low-level strength number.
6. Perform security test and proactive attacking.
7. Increment the protection strength level if security test failed.
8. Repeat 5-7 until you get acceptable result.

White the protection value provides an analytical representation of the security level; it is only to give the designer/developer confidence to come up with "*good enough protection.*" The most default and subjective steps in this analysis is security testing and proactive attacking. While this could potentially be an open-ended practice, there should be a protection value that is deemed good enough by designer/developer.

5.4. Performance Versus Security

Typical cryptographic operations are time and resource constrained and they must be measured, adjusted, and optimized. There is always an additional overhead associated with software implementations of cryptographic functions as oppose to hardware variant. The concept of "*good enough protection*" has another angle that is to create a balance between application performance and security. This is particularly apparent in applications with time sensitive functions in real time. For instance, over-the-top (OTT) video delivery applications must process, decrypt, decode, and render, video data in real time while maintaining the smooth playback experience. Typically, the entire video-handling module of this application must be secured to not expose valuable video and audio data during delivery. At the same time, the module must

perform fast enough to provide a non-interrupted play experience. That requires a delicate balance between security and performance.

Designing and implementing a "*tunable*" mechanism for both performance and security within the application can achieve this. Here are some considerations to be aware of in designing tunable security functions:

- The performance of the crypto algorithm
- The overhead of encrypt and decrypt operations in real time
- The overhead of software obfuscation
- The memory intense computation of white-box cryptography operations

5.5. Security Analysis Flow

Security analysis is an ongoing and iterative process that must be performed parallel to the normal software development cycle. Every time a new feature is introduced, developed and tested, its protection needs to be evaluated. Policies and practices are evaluated during the design phase from a security point of view. Then *static analysis* is performed by each developer per each development cycle to identify any leaking security holes. The process continues into the application execution phase where *dynamic analysis* and *penetration testing* are conducted in the runtime. The security analysis flow phases and details are shown below.

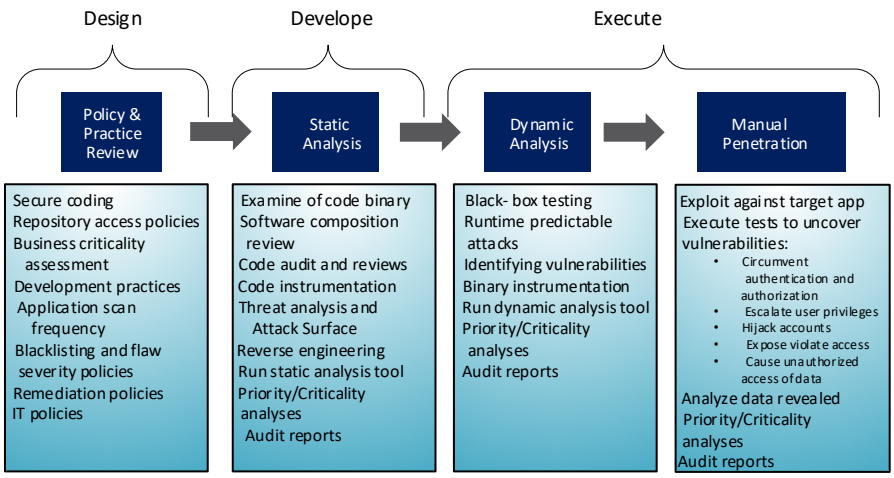

Figure 16 - Security Analysis Flow

The static and dynamic analyses are an essential part of this process to help developers quickly identify and remediate application security flows every time a new feature is introduced [32].

6. Protection Guideline and Best Practices

6.1. Practical Guideline to Protection

A typical software application consists of high-level user interfaces interacting to commands from a user or with other applications to low-level system interfaces making use of platform specific features. While

these interactions could be implemented in various modules in different layers, there is always data exchanges and handshaking to be done among them.

An effective way to ensure comprehensive application protection is to examine and apply protection in all layers and in both bottom-up and top-down directions. While the principal security consideration remains the same, the attack surfaces might be different in each layer. In bottom-up, a designer/developer starts with exploring vulnerabilities of specific deployment platform and works their way up to the possible security holes in the application interfaces. Application development platform tools including TEE providers are typically in charge of ensuring low-level module security. Thus, assuring any application written on those platforms comes with certain level of protection out-of-the-box. Whereas in the top-down approach, the designer/developer starts with user interfaces and works all the way down to platform interactions.

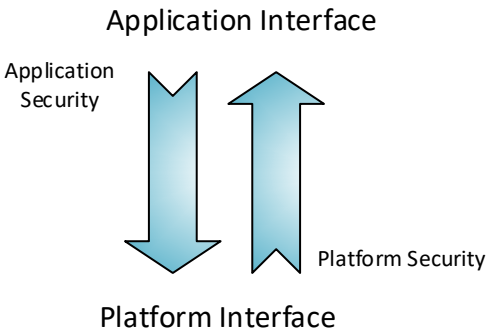

Figure 17 - Top-Down and Bottom-Up Security Approach

All levels in between are examined for vulnerabilities. Therefore, both approaches shall be included in the protection guideline.

6.2. Security Everywhere!

Application protection is a never ending proposition. This means there will be always a hidden vulnerability somewhere in the software application to be discovered given time and effort. Many good security designs do not deliver good security, not because the defensive theory is unsound, but because the designers cannot achieve integration into real-work systems [33]. This is also true for any inner modules of application as they need to be integrated internally.

6.3. Deep Preventive Measures

While building a multi-layered security and defense mechanism within the application is a key aspect of application protection, it is equally important to "fail intelligently" when an attack occurs. If the application fails for whatever reason, it should not expose any sensitive data, or give any detail to hint the attackers about where it failed. This is hard to achieve yet doable to a certain extent knowing that attacks are inevitable. Applications should be built with pre-defined, but not predictable failures in the face of an attack. For instance, once the anti-debug detection is triggered, there should be a randomized and unpredictable application failure without any trace. In some cases, it may be acceptable to crash the application when such an attack has occurred. Designers/developers should also consider implementing

deep preventive measures where an application doesn't continue execution if certain events (attacks) are detected.

6.4. Best Practices for Application Protection

There are many organizations, forums, consortiums, government agencies, and independent individuals providing best security practices for various types of applications [34] [31] [35] [36].

Exploring those best practices are outside the scope of this paper. However, it is important to pay special attentions to the following items, while considering these best practices:

- Sensitive data and metadata at rest (cached or stored)
- Sensitive data and metadata in transit (to/from application)
- Application logs, crash logs, debug symbols, and tokens with no traceable channels
- Critical and secretive code
- Single point of failure and simple logic to make important decisions
- Copy/paste capabilities
- Third-party libraries with known vulnerabilities
- Exposure of geolocation data and routines
- Memory scrambling capabilities of the platform

7. Application Protection Cases

A strong white-box crypto combined with obfuscation and other protection techniques discussed in this paper opens a great potential with a lot of flexibilities to designer/developer in protecting their applications. Here we highlight some of these cases.

7.1. Cloud Video Distribution

As more companies are transitioning to cloud-based video delivery system, they are relying on the security of the public cloud provider ecosystem such as AWS and Google Cloud. While there is no question that public cloud providers have done their due diligence to make sure their core cloud offerings are secured, analyzing their security capabilities are beyond the scope of this paper. Most of them offer a form of Cloud Hardware Security Module (CloudHSM) that brings hardware security strength to cloud application for a fee.

These services can be very costly for video delivery systems in the cloud since most of these services are usage-based which may not be feasible with the typical large volume of video data transactions. These cloud-based services process cryptographic operations and provide secure storage for cryptographic keys.

Whether to use CloudHSM or have all software-based security in the cloud, the protection of video delivery from streaming applications remains the core responsibility of the application owner. In an end-to-end video distribution system, attacks are often aimed at defeating conditional access or finding ways of exploiting services that are easier than circumventing cryptographic protection. A strong white-box

crypto combined with obfuscation and other protection techniques should provide an alternative protection to using costly hardware-based solution for cloud video delivery applications.

7.2. Digital Rights Management

Digital Rights Management (DRM) applications are another candidate for all software-based protection. While hardware-based security might be still needed when hardware root of trust is used, all other crypto operations can be done in the white-box. Authentication, authorization, right verification, key exchanges, and more can utilize software obfuscation with the additional protections discussed in this paper.

7.3. SSL/TLS Services

OpenSSL is a widely used proven technology to implement Secure Socket Layer (SSL) and Transport Layer Security (TLS) for application. The security promise of these protocols is to protect information while in transit over the network. There is no guarantee or provision within these protocols to protect information that resides on either end of these secure pipelines, including the secrecy of the very keys upon which these protocols depend. To fill in the gap, white-box crypto can provide a viable and secure alternative by seamlessly replacing underlying cryptographic operations in OpenSSL while maintaining compatibility with the existing API. This will enable secure handling of keys and other sensitive cryptographic assets without the need to significantly alter existing applications that utilize OpenSSL.

7.4. Public Key Infrastructure

Public Key Infrastructure (PKI) applications can also utilize software-based protection ranging from servicer identification certificate delivery to generating application specific keys. To add further security, protected keys can be locked to a host machine, container, or application such that even if an external party gains access to them (say through an insecure application running inside the same container), the encoded keys are not usable outside of that container.

7.5. Internet of Things

Internet of Things (IoT) devices typically don't have sophisticated hardware capable to support hardware-based security or TEE. This makes it a perfect candidate to use all software-based protection with some light-weight white-box and obfuscation. This truly enables IoT providers to deploy their application in scale that was never possible with systems depending on hardware-security or with no security designed into the software.

Conclusion

In this paper, we discussed various software protection technologies that can fill in the gap in securing applications without the use of hardware security. Recent technological advancements in white-box cryptography and software obfuscation makes it feasible to deploy software-based protected applications that are flexible and easier to maintain. The holistic approach to security allows developers to view application protection from various angles. This enables companies to deploy iterative security analysis processes in their software development practices, utilizing feedback received from attack surface analysis and penetration test results. There are all kinds of applications in various domains with different

use cases that can benefit from these technologies either by creating an entirely new protection scheme or enhancing existing security mechanisms.

Abbreviations

API	Application programming interface
bps	Bits per second
AES	Advanced Encryption Standard
CloudHSM	Cloud Hardware Security Module
DRM	Digital rights management
ECC	Elliptic-curve cryptography
IDE	Integrated development environment
IoT	Internet of things
ISBE	International Society of Broadband Experts
OEM	Original equipment manufacturer
OWASP	Open Web Application Security Project
PKI	Public key infrastructure
RSA	Rivest–Shamir–Adleman
SCTE	Society of Cable Telecommunications Engineers
SSL	Secure sockets layer
TA	Trusted application
TEE	Trusted execution environment
TLS	Transport layer security

References

[1] TechTarget, *What is holistic security? - Definition from WhatIs.com.*

[2] B. Schneier, "Tales from the Crypt: Hardware vs Software," 2015.

[3] H. Bar-El, "Security implications of hardware vs. software cryptographic modules," pp. 1-3, 2002.

[4] S. Chow, P. Eisen, H. Johnson and P. C. Van Oorschot, "A white-box DES implementation for DRM applications," pp. 1-15, 2003.

[5] L. A. Anderson and S. Medvinski, "Candidate fully homomorphic white-box construction," 2016.

[6] C. Delerablée, T. Lepoint, P. Paillier and M. Rivain, "White-box security notions for symmetric encryption schemes," vol. 8282 LNCS, pp. 247-264, 2014.

[7] S. D. Galbraith and L. A. Anderson, "White-box cryptography," pp. 1-12, 2013.

[8] O. Billet, H. Gilbert, C. Ech-Chatbi and Springer, "Cryptanalysis of a white box AES implementation," vol. 3357, pp. 227-240, 2005.

[9] Y. De Mulder, B. Wyseur and B. Preneel, "Cryptanalysis of a perturbated white-box AES implementation," 2010.

[10] T. Lepoint and M. Rivain, "Another Nail in the Coffin of White-Box AES Implementations," 2013.

[11] J. W. Bos, C. Hubain, W. Michiels and P. Teuwen, "Differential Computation Analysis : Hiding your White-Box Designs is Not Enough," pp. 1-22, 2015.

[12] L. Goubin, P. Paillier, M. Rivain and J. Wang, "How to Reveal the Secrets of an Obscure White-Box Implementation," no. 643161, pp. 1-22, 2018.

[13] S. Chow, P. Eisen, H. Johnson and P. C. Van Oorschot, "White-Box Cryptography and an AES Implementation," pp. 250-270, 2003.

[14] P. Fouque, P. Karpman, P. Kirchner, B. Minaud, B. M. Efficient and P. White-, "Efficient and Provable White-Box Primitives," 2017.

[15] L. A. Anderson, "White-box node-locking," pp. 1-20, 2015.

[16] A. Biryukov, C. Bouillaguet and D. Khovratovich, "Cryptographic schemes based on the ASASA structure: Black-box, white-box, and public-key," 2014.

[17] A. Bogdanov and T. Isobe, "White-Box Cryptography Revisited: Space-Hard Ciphers," pp. 1058-1069, 2015.

[18] B. Wyseur, "White-Box Cryptography," no. May, pp. 1-9, 2008.

[19] L. A. Anderson, *White Box-Development Guide,* 2015.

[20] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan and K. Yang, "On the (im)possibility of obfuscating programs," vol. 59, no. 2, pp. 1-48, apr 2001.

[21] S. Chow, Y. Gu, H. Johnson and V. A. Zakharov, "An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs," pp. 144-155, 2001.

[22] L. A. Anderson, "A survey of control-flow obfuscation methods," no. October, 2015.

[23] L. A. Anderson, *Candidate randomized branch-encoding primitive,* 2014.

[24] F. Kitagawa, R. Nishimaki and K. Tanaka, "Simple and Generic Constructions of Succinct Functional Encryption," vol. 10769 LNCS, pp. 187-217, 2018.

[25] M. N. Garnon, S. Taylor and A. K. Ghosh, "Software protection through anti-debugging," vol. 5, no. 3, pp. 82-84, 2007.

[26] Apple Inc, *iOS Developer Library,* 2016.

[27] Microsoft, *Microsoft Developer Network,* 2016.

[28] T. Schwarz and S. J. COEN, *Santa Clara University,* 2004.

[29] L. A. Anderson, "Dynamic Executable Verification (DEV)," 2016.

[30] P. Larsen, A. Homescu, S. Brunthaler and M. Franz, "SoK: Automated software diversity," pp. 276-291, 2014.

[31] OWASP, *OWASP Cloud Security Project.*

[32] SCTE, *SCTE IoT Security Best Practices,* 2017.

[33] Akamai, *Web application firewall - A champions guide,* 2018.

[34] NowSecure, *The Mobile App Security Company.*

[35] Black Hat, *Black Hat USA 2015 | Briefings.*

[36] EC-Council, *iClass Certified Ethical Hacker - InfoSec Training.*

[37] Y. De Mulder, P. Roelse and B. Preneel, "Revisiting the BGE Attack on a White-Box AES Implementation," 2013.

[38] OWASP, *OWASP Foundation.*

