intertrust®

INTERTRUST SECURE SYSTEMS WHITE PAPER

# Application Shielding with Intertrust's

# whiteCryption® Code Protection™

CPWP082018

# TABLE OF CONTENTS

# Executive Summary

The world of computing has changed. Security is not just about physically secure data centers and corporate controlled computing assets. Instead, end users have gone mobile, connecting to cloud enabled services, often with their own personal devices. And with the rise of the Internet of Things, there will be billions of connected computing devices on the planet in the next several years. These changes significantly impact the way organizations are providing services to their customers, enabling new business models and new ways to do business.

But these changes also create new opportunities for hackers who have unprecedented physical access to these devices. Hackers have a wide variety of goals including bypassing business logic, stealing intellectual property or sensitive data, obtaining cryptographic keys to steal content, masquerade as users, snoop on secure communications, or hack a device as a stepping stone to launch attacks against other devices. If they are successful, then the consequences can include financial loss, impact on brand reputation, and exposure to liability.

The way hackers go about achieving their goals starts with reverse engineering software to find vulnerabilities they can exploit, data they can extract, or ways to modify the software to do something it was never intended to do. As hackers get increasing access to mobile and IoT devices, this threat also increases. As a consequence, it is becoming increasingly important for organizations to deploy **application shielding** technology that makes it difficult for hackers to reverse engineer and tamper with software.

This white paper describes these threats in detail, and describes Intertrust's market leading application shielding product, **whiteCryption® Code Protection™**. Code Protection provides application developers with a comprehensive suite of anti-reverse engineering and runtime application security protections to help protect your applications. Code Protection is easy to use and requires no significant changes to the code itself or the existing build chain. Since Code Protection secures source code before it is compiled, protected builds can easily be delivered to an app store, end point, mobile device, connected car, and other types of IoT devices.

# 1 The Problem

Computer and information security is one of the biggest problems that businesses face today. According to a recent SANS Institute study, organizations spend as much as 12% of their IT budget on security[1]. In a Ponemon Institute study, it was found that organizations have a 27.7% probability of having a material data breach in the next 24 months at an average cost of $3.62M[2]. There are many aspects to security, ranging from hardware security, application security, and data security, all the way through governance and risk management. The focus of this white paper is **application security**.

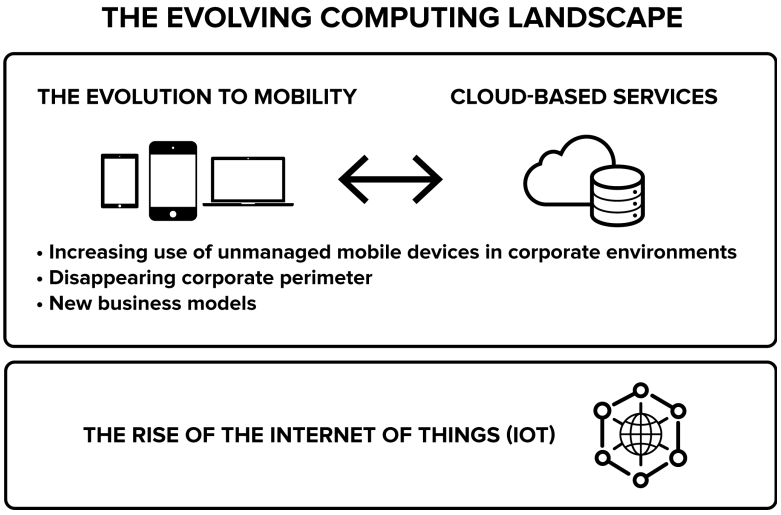
The primary consequences of applications getting hacked include financial loss, destroyed brand reputation, exposure to liability, and regulatory risk. Over 7 billion identities have been stolen in data breaches over the last eight years equal to one data breach for every person on the planet[3]. Meanwhile, mobile's rapid expansion has introduced a complicated and potentially hostile environment that is difficult to manage and protect. 64% of security practitioners said they were very concerned about the use of insecure mobile applications in the workplace with an average of 472 mobile applications reported as actively used in organizations[4].

## 1.1 The Evolving Computing Landscape

There are two major trends in computing that impact application security. The **first** is that computing is evolving towards highly mobile devices connected to cloud-enabled services. The **second** is the advent of the Internet of Things (IoT).

Both of these trends mean hackers have more opportunities to have direct physical access to applications, whether those applications are on mobile or IoT devices, allowing them a significantly larger attack surface.

We discuss each of these trends in more detail below.




### 1.1.1 The Shift to Mobile Devices and Cloud Enabled Services

Prior to the advent of mobile computing, security was limited to corporate IT assets that were often physically secured in facilities owned and managed by the company, on a network behind a managed firewall, and possibly in a datacenter with multi-factor access, physical security, and armed guards. Because the company owned those assets, they were able to dictate what applications could run on those machines, and actively manage and monitor them, providing the latest patches, endpoint security, and other controls dictated by corporate IT.

Assets located in such places were implicitly trusted.

Today, the situation has changed. Mobile devices dominate the market[5], often as the primary or only way users access the Internet and the many cloud services available. These smartphones and tablets are often individually owned, and as a result are largely unmanaged. They are used for both personal and work activities. Employees use those devices to engage in risky behavior that would normally not be tolerated in a managed corporate setting, exposing organizations to many types of risk including malware infections that spread to the corporate network. These devices also have very little, if any, physical security. All of these realities create a fertile field for hackers trying to exploit code and socially engineer the result they desire. It is a well-worn path hackers use to access such devices to reverse engineer or tamper with the applications running on them, often through rooting, jailbreaking or hoodwinking the user.

This shift has created all sorts of new business models to take advantage of the popularity of mobile devices. These new business models come with new security problems. For example, new forms of payment using near field communications (NFC) on mobile devices are becoming popular in recent years. These applications require that *credentials to authenticate users must be stored on the device*. If those credentials are compromised, then a hacker can execute fraudulent transactions. Mobile devices are being used in the automotive industry to enable remote parking from your smartphone. A compromise of the device could pose a serious safety risk. In healthcare, patients are using mobile devices to manage sensitive information collected from various devices ranging from fitness monitors to blood glucose monitors to improve care and create data driven treatment options. A compromise of such a device can lead to a loss of privacy and sensitive information. Or even worse, if a device is hacked, it could potentially lead to life-threatening consequences for the patient.

### 1.1.2 The Coming Revolution of the Internet of Things

Computing and communication have become so inexpensive and commoditized that network enabled computers can now be embedded in almost everything, from thermostats in your home to wind turbines. IoT is yet another example of computing devices in largely physically insecure and often unmanaged environments. Since we can't lock down and control these systems like we did with mainframes, servers and PCs, we must use newer tools and solutions to solve these problems.

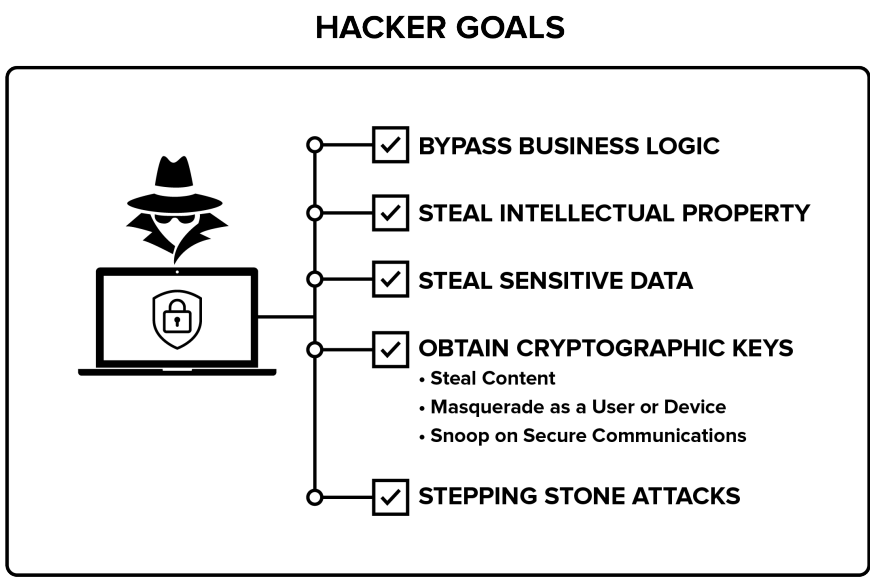
The Internet of Things exponentially increases the number of devices behind a network's firewall with cars, homes, wearables and many other devices allowing hackers to accomplish more, quicker. Even still, IoT application developers *continue to put emphasis on user-end convenience over security*. 68% of survey respondents say end-user convenience is very important while only 30% say their organization allocates sufficient budget to protect mobile apps and IoT devices[6].

By 2025, the total global worth of IoT technology will reach USD 6.2 trillion with the most value coming from health care devices (USD 2.5 trillion) and manufacturing (USD 2.3 trillion)[7]. Meanwhile, we see a persistent lack of IoT security investment with 67% of medical device makers expecting an attack on their devices while only 17% taking measures to prevent an attack[8]. These numbers are staggering when you consider U.S. hospitals have an average of 10 to 15 connected devices per bed with some hospitals registering 5,000 beds – totaling 50,000 connected devices per hospital.

Furthermore, traditional security solutions do not port well to the IoT world, due to differences in system architectures and resource constraints. Therefore, IoT security solutions have not evolved enough and are prone to numerous vulnerabilities[9].

## 1.2 What Are Hackers Trying to Achieve?

In order to understand threats, we must understand what hackers are trying to achieve. Hackers will mount different kinds of attacks to achieve different kinds of goals. And so, defending against hackers in the context of application security may involve defending against many different kinds of attacks.




Hackers might be interested in bypassing business logic. For example, they might want to bypass controls that let them cheat at a video game or violate the terms of a software license. Of more serious concern is the potential for hackers to bypass controls in safety critical systems. It is not inconceivable that lives could be at risk if a hacker were able to hack a medical device, connected car or some component of critical infrastructure, such as a wind farm[10], a coal or nuclear power plant, a power grid, or a water treatment facility.

Many organizations spend millions of dollars developing applications that contain millions of lines of code. According to a recent study, automobiles today run systems that have more than 100 million lines of code[11]. Those applications often contain valuable intellectual property, which hackers would rather steal than develop. For example, they might be a competitor (or nation state) with inferior technology attempting to improve their own products in order to compete more effectively.

Hackers might also be interested in obtaining valuable pieces of data that are managed within the application, such as music or video, financial data, or privacy sensitive health data.

While data can be protected with cryptography, this only shifts the problem from protecting the data directly to protecting the *cryptographic keys*. Cryptographic keys are not only used to protect data. They can also be used to create a secure identity for a device. A device may need such a key to authenticate to a cloud service. If a hacker were able to obtain this secret, they might be able to masquerade as that device or as the owner of the device. Cryptographic keys are also used to establish secure communications. For example, HTTPS is a familiar protocol that uses SSL/TLS to secure communication to websites. If a hacker were able to obtain these keys, they could snoop on or alter supposedly secure communications.

For all of these reasons, hackers are highly motivated to steal cryptographic keys embedded in or controlled by an application.

Finally, sometimes hackers aren't interested in the application itself, but using the application as a digital stepping stone to try to achieve some other goal. Often hackers are interested in obtaining root access on the device the application is running on, so they can install malware or use the device as a launch pad to attack something else.

Consider the 2016 Mirai botnet that infected web enabled cameras and installed a piece of malware that launched the largest distributed denial of service attack[12] in history against the dynamic domain name service Dyn, causing wide spread internet outages. Those 100,000 cameras were able to launch 1.2 terabytes per second of data at a major piece of the global internet infrastructure. Here the goal of the attacker was not to compromise the webcam directly, but rather to bring down the web services of many companies whose DNS was controlled by Dyn.

## 1.3 How Hackers Work

Hackers employ two fundamental techniques when attacking: **reverse engineering** and **tampering**.

If the hacker is trying to bypass business logic, they have to find where in the application the business logic resides. That requires reverse engineering. Then they typically must tamper with the application to bypass that logic.

If the hacker is trying to steal intellectual property, sensitive data or cryptographic keys from an application, they have to know where to look in the application. Unless those secrets are obvious, hackers need to reverse engineer the application to find them.

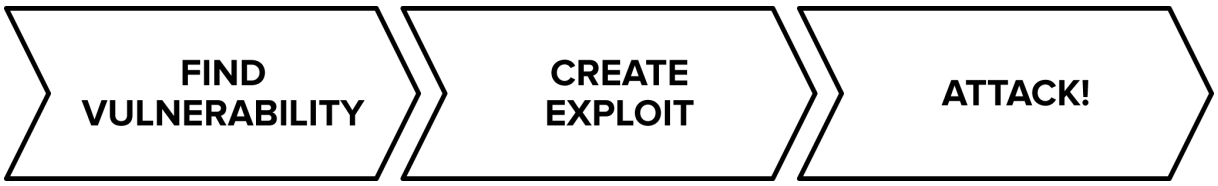



**Figure 1: Active attack workflow**

If the hacker is trying to create a stepping stone attack, they often use the workflow shown in Figure 1. First, they find some vulnerability in the application, which again requires reverse engineering. Then, they craft an exploit that takes advantage of that vulnerability. Finally, they attack by launching the exploit to the application. In a remote attack like the popular SQL injection attack, this may involve sending the message to the application over the internet. But if they have physical access to the device, which with mobile and IoT based systems can be as easy as a trip to the store, then they can directly tamper with the device.

You must expand your threat matrix into this new reality.

## 1.4 Why Can't We Just Get Rid of Vulnerabilities?

An obvious observation from Figure 1, is that if the application has no vulnerabilities, then the hacker would be stopped in his tracks. In an ideal world, we'd eliminate all vulnerabilities and the security problem would largely be solved, and we could all go home (at least those of us in the security industry).

Sadly, the reality is that vulnerabilities are not going away any time soon.

There is no shortage of vulnerabilities. At last count, the National Vulnerability Database managed by NIST contained over 88,992 vulnerabilities[13]. Most large product development organizations have dedicated teams that track vulnerabilities and issue alerts accordingly. Even the most diligent of development teams still produce applications with vulnerabilities.

Eliminating vulnerabilities appears to be a fundamentally difficult problem to solve. Even the best developers write code with bugs. Studies have shown that most software has 15-50 bugs[14] per 1000 lines of code. Not all bugs are security vulnerabilities, but many are. It is not unusual for modern applications to contain millions of lines of code, meaning thousands of potential vulnerabilities to be exploited. This simple math means it simply impossible to create vulnerability-free code.

Vulnerabilities are not just due to bugs. Design flaws can be exploited, too. While some design flaws are just due to poor security considerations during the design process, there are many design flaws that are subtle and can go unnoticed for years, like the Heartbleed attack. This attack on OpenSSL relied on exploiting a heartbeat mechanism built into the SSL protocol and it was in the wild for at least two years, although how long it was exploited is unknown[15].

Certainly, we want to identify **known** vulnerabilities in software and eliminate them. There is great value in application security testing tools to help with exactly this problem. Many vendors provide such tools. They play an important role in a secure development lifecycle. But, the discovery of a vulnerability in an application is just the first step toward mitigation. A patch needs to be created to fix the vulnerability, then that patch must be deployed. All of this takes time. According to a WhiteHat Security study[16], it takes an average of 150 days fix vulnerabilities, with some industries taking significantly longer. Moreover, even if you have a patch, finding all of the systems with that vulnerability might not be possible.

***"Application shielding can play an important role in hiding vulnerabilities that exist but are not yet known, making it difficult for hackers to achieve their objectives."***

But, we can only fix the vulnerabilities that we know about. Arguably, the most dangerous vulnerabilities are those which are **unknown**, i.e. those that have not yet been discovered. You can't create and deploy a patch for a vulnerability that you don't know about.

The reality is that vulnerabilities are always going to be there whether you know about them or not. While it's preferable to eliminate vulnerabilities entirely, application shielding can play an important role in hiding vulnerabilities that exist but are not yet known, making it difficult for hackers to achieve their objectives.

## 1.5 Reverse Engineering

Reverse engineering plays a central role to almost every hacker when attacking applications.

Reverse engineering is extraordinarily hard work. The people who do it have a large toolset and focused mind. Even understanding well written software can be a challenge. Software engineers are taught to create design documents and functional specifications, insert plenty of comments, and both have and use defined coding standards. They are taught these practices for a reason: to make the code as understandable as possible to minimize bugs so that other programmers can maintain and upgrade code as efficiently as possible. Still, even well written, well documented code can be a puzzle to understand.

But the hacker's job is immediately more difficult because the source code has been compiled into assembly language or virtual machine instructions. A lot of information is lost during the compilation process. There is no documentation, comments, or coding standards that the hacker can rely on to help them understand the code.

Nevertheless, hackers are smart and tenacious and will spend copious amounts of time reverse engineering software in order to achieve their goals. Because their goals are totally different from the goals of software engineers, their techniques are different, too. For example, they are not concerned with fixing bugs; in fact, it is common to *introduce artificial bugs* into the compiled code to see how it breaks. Instead, they are focused on achieving a very specific goal, and will look for the exact places in the compiled code that they need to understand in order to achieve their goals and not worry about anything else.

They are familiar with common structures in compiled code. For example, they might look for a string corresponding to an error message related to their objective (e.g. "permission denied") and trace where that string is used. They leverage sophisticated techniques such as static analysis, which help them understand the overall structure of the code, where the functions are located, how they are called from other functions. Talented reverse engineers can look at assembly language code and immediately recognize that it is, for example, performing cryptographic operations.

## 1.6 Tampering with Code

If hackers have physical access to a device they can directly tamper with the application. Consider how you might bypass business logic. Often, the way this is done is by making one small change to the application. Consider the following code:

```
if (hasPermission(user, resource, action)) {
     grantAccess(user, resource, action);
} else {
     denyAccess(user, resource, action);
}
```

If the hacker is able to identify this particular piece of code, then they can often bypass it by any number of ways, typically at the assembly language level:

- Inverting the logic of the conditional jump

```
if (!hasPermission(user, resource, action)) {
      grantAccess(user, resource, action);
} else {
      denyAccess(user, resource, action);
}
```

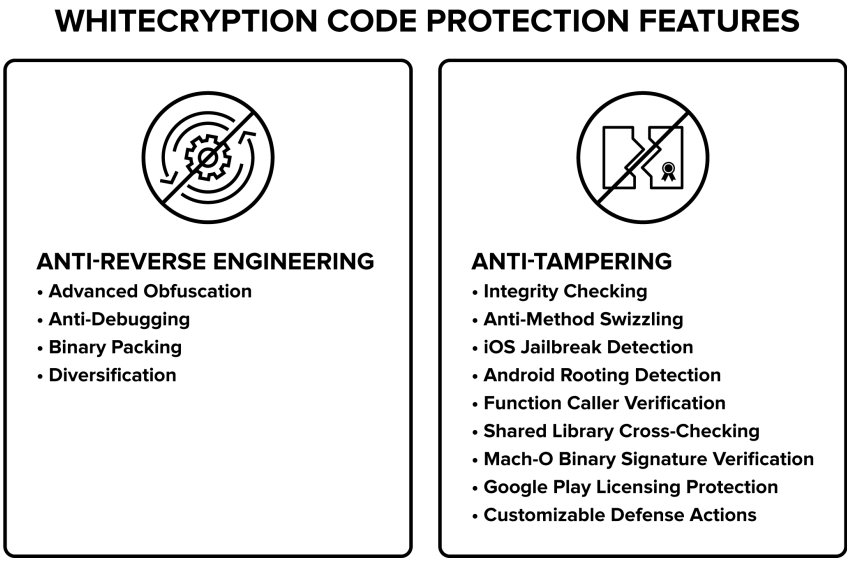
- Replacing the test with a tautology

```
if (true) {
      grantAccess(user, resource, action);
} else {
      denyAccess(user, resource, action);
}
```

They might also replace function calls to functions of their own design. For example, they might trick the application into calling their own version of `hasPermission()`, which grants permission to exactly what they are after.

# 2 The Solution

In the previous section, we showed how reverse engineering and tampering with code are fundamental tools hackers use to achieve their objectives. We also showed how hackers have more opportunities to have physical access to devices because of the explosion in mobile devices, new business models, and the Internet of Things. These trends mean hackers have more opportunities to have direct physical access to applications, where the hacker can reverse engineer the code to steal intellectual property or sensitive data, can tamper with the application to bypass business logic or create stepping stone attacks.




The solution to these problems lies with **application shielding**, which prevents reverse engineering and tampering with code.

**Intertrust's whiteCryption Code Protection** provides the industry's best application shielding solution available on the market.

**Code Protection** is a comprehensive code protection solution intended for hardening software applications on multiple target platforms. It adds tamper resistant characteristics to applications by applying code obfuscation, integrity protection, anti-debug, and anti-piracy techniques to application code (see Figure 2).

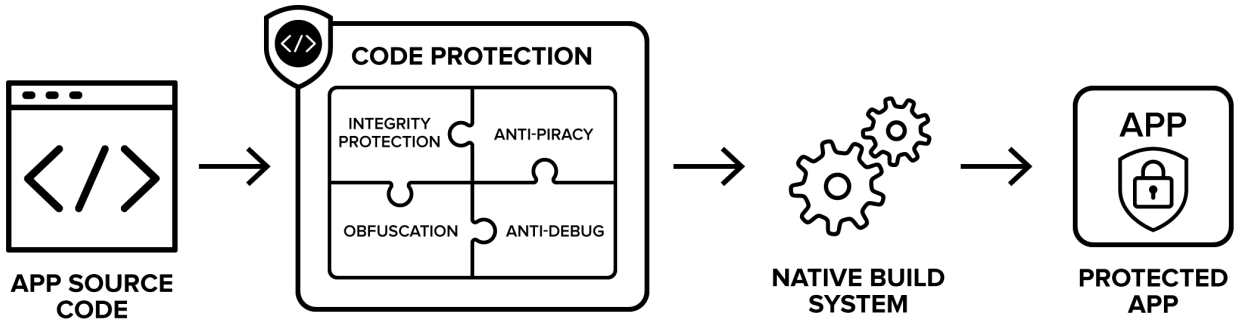



**Figure 2: Intertrust's whiteCryption Code Protection Overview**

Code Protection can protect any standards compliant C/C++/Objective-C/Swift or Android Java source code **and requires no significant changes to the code itself or the existing build chain**. Most of the security features are automatically added to the application, and the configuration of individual features and security strength is easily accomplished using the intuitive graphical user interface. All functions are available through command line for integration into automated build systems.

A highly specialized form of anti-reverse engineering is **whitebox cryptography**, in which a special cryptographic library is provided that provides strong protection for cryptographic keys. In whitebox cryptography, the underlying mathematics of the cryptographic operations are obfuscated in such a way that the keys never appear in the clear. Standard operations such as encryption, decryption, secure key unwrap and digital signature creation and validation can all be done with whitebox cryptography techniques, protecting the keys even if the device is jailbroken or rooted.

**Intertrust's whiteCryption® Secure Key Box™ (SKB)** provides an industry leading whitebox cryptography solution to this problem and protects against modern side-channel attacks like DFA/DCA. A full discussion of whitebox cryptography is beyond the scope of this paper. For a detailed overview of Secure Key Box, see our website and companion white paper.

## 2.1 Anti-Reverse Engineering

By making code difficult to reverse engineer, hackers will have a harder time finding vulnerabilities. Thus, application shielding not only protects against stealing intellectual property or sensitive data, but it can also protect your application by making it hard for hackers to find vulnerabilities.

Our customers's objective is to protect their applications for as long as possible against all threats. We enable this by make the hackers' job of reverse engineering code as difficult as possible.

**Intertrust's whiteCryption Code Protection** provides the industry's best anti-reverse engineering functionality.

### 2.1.1 Advanced Obfuscation

One of the key techniques used in anti-reverse engineering is **code obfuscation**. The term "obfuscate" means to render obscure, unclear or unintelligible. The goal is to remove as much of the structure as possible that would be familiar to reverse engineers, to make the code as confusing as possible while keeping functionality the same.

**Control flow obfuscation** modifies the basic structure of how subroutines are called. For example, calls to subroutines could be replaced with computed jumps and functions can be inlined. Symbols for function names can be replaced with random strings. For example, in the Objective-C programming language, commonly used on Apple platforms, the function names are generally preserved after compilation, which gives hackers a valuable piece of information about the structure of the code. We provide a feature to encrypt this Objective-C Metadata to further frustrate hacker efforts. Dummy blocks that serve no purpose other than leading the hacker down a dead end are also frequently used.

**Intertrust whiteCryption Code Protection** is capable of inlining static void functions with simple declarations into the calling functions. Such operation increases the obfuscation level of the final protected code and makes it more difficult to trace. The overall result is increased security of the protected application.

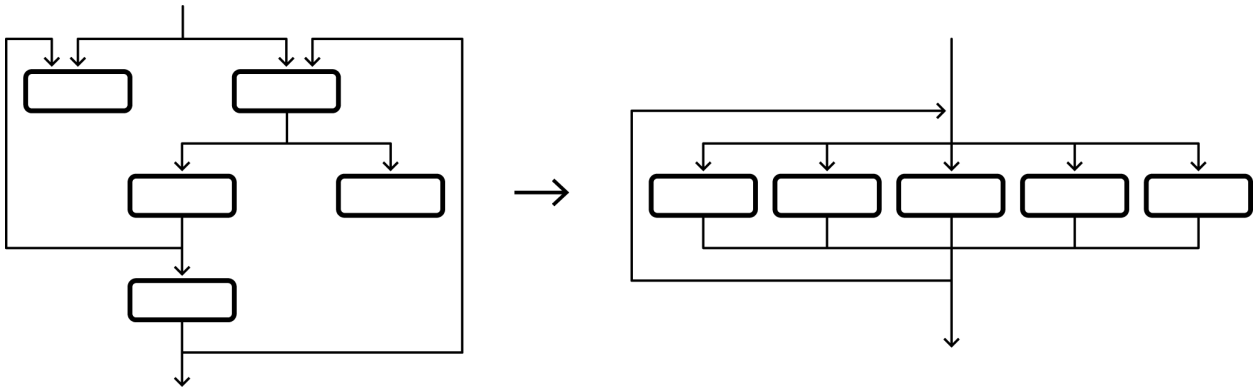


**Figure 3: Control flow flattening**

Another important technique is **control flow flattening**, in which routines are not called directly from other routines, bur rather a dispatcher controls the control flow as illustrated in Figure 3.

Most code is made up of a series of basic blocks, which are some set of non-branching instructions (arithmetic or logical operators) followed by a conditional or absolute jump. The structure of a function can be viewed as a graph of basic blocks, kind of like a flow chart. Basic blocks can be rearranged, split or combined. For example, the basic blocks of all the functions can be intermingled, so that the code for any one function is not contained in a contiguous region of the executable.

An absolute jump can be replaced with a computed jump adding edges to the graph. For example, consider the following code fragment:

```
if ((x*x) & 2) {
      foo();
} else {
      bar();
}
```

This seems quite straightforward. It takes the square of the number `x` and branches to function `foo()` if the second least significant bit is one, and to `bar()` if that bit is zero. However, it turns out that the second least significant bit of the square of any number is always zero[17], thus `foo()` will never be called. Why would we do that? We could replace `foo()` with another real function in the code, one that looks important. Static analysis tools cannot generally figure out that this jump will never be taken, and so the attacker may spend valuable time trying to understand what this part of the code is doing.

Dummy basic blocks can be introduced increasing the size of the graph, and thus the complexity of understanding the code.

Imagine inserting thousands or even millions of such indirections, creating a tangle of function calls resulting in epic hacker frustration. It's a good feeling.

Another feature to obfuscation control flow is **Objective-C Message Call Obfuscation**. Objective-C is designed so that messages to object instances are resolved only at run time. Because of this fact, message calls are stored in plain form in the binary code. This is an attack vector that hackers can use to manipulate the execution logic. Code Protection provides a powerful security feature that obfuscates message calls in the binary code, thus making reverse engineering more difficult.

**Intertrust's whiteCryption Code Protection** provides a security feature that obfuscates a large portion of string literals (including Objective-C string literals, which are NSString pointers) in the code and deobfuscates them only before they are actually used. This feature increases protection against static analysis.

Finally, **metadata obfuscation** is also important. Executable files often come with metadata in the headers that can provide useful information to hackers about how to reverse engineer code.

An Objective-C executable contains metadata that provides information about names of classes and methods, as well as method arguments and their types within the executable. This information can aid attackers when they are statically analyzing the program with a disassembler.

Code Protection can encrypt some of the Objective-C metadata to partially hide the useful information from static analysis tools. The encrypted metadata is only decrypted at run time when it is used by the protected application.

### 2.1.2 Anti-Debugging

One of the primary tools hackers use to reverse engineer code is debuggers. Normally used by legitimate software engineers to find bugs in code, debuggers give hackers a powerful tool from which to reverse engineer code. Debuggers generally work by setting interrupts at specific points in an executable, thus modifying the executable. When the instruction counter reaches that point in the code, an interrupt is raised and the program stops at that point. Debuggers like IDA Pro do other things like disassembly and decompilation.

Code Protection inserts numerous anti-debug checks into your protected application. These checks take into account the unique indications of the target platform that may identify the presence of a debugger. When a debugger is detected, an appropriate defense can be taken (see section 2.2.9).

These anti-debug checks use kernel syscalls, thereby bypassing usermode hooks the hacker may have inserted. This forces an attacker to modify their kernel, which significantly raises the bar for a successful attack.

### 2.1.3 Binary Packing

**Binary packing** is a trick that hackers use themselves when infecting a platform with malware. The code being downloaded is encrypted and unpacked at runtime. This makes it hard for endpoint detection to find malware because the malware is encrypted with a different key each time so there are no recognizable patterns in the packed code for endpoint protection systems to detect.

With **Intertrust whiteCryption Code Protection** the binary version of the compiled code is encrypted, and a stub is inserted that decrypts it into memory only at run time. This makes it difficult for hackers to reverse engineer. They can't run the code through a disassembler or decompiler. Instead, they must capture the code after it has been decrypted into memory and only then reverse engineer it. This virtually stops all static analysis, forcing the attacker to apply more complex and time consuming dynamic analysis techniques.

Binary packing encrypts the code and read-only data sections of .so libraries generated from application C/C++ code as well as native C code automatically generated by Java Code Protection. Thus, it effectively provides strong protection against static code analysis for both the native C/C++ and Java portions of an Android application.

### 2.1.4 Diversification

A common challenge with application security is that if a hacker can successfully break one instance of an application, then they might be able to create an automated tool that can break any other instance of the application. This is known as the **break once, run everywhere** (BORE) problem.

The obfuscation techniques discussed above can be done using randomization. As a result, it is possible to create instances of code that are different from one another, thwarting break once, run anywhere attacks. This technique is known as **diversification**, and can be a powerful tool in protecting deployments of applications. Diversification can be done across a population of applications, so that conceivably every application instance is different. But it is also a useful technique for diversification of versions of code. For example, if a hacker successfully reverse engineers Version 2.3 of an application, they will have to start all over again once Version 2.4 is released. Updated code is functionally similar but the surface of the code is unique and dramatically different in shape and structure.

Diversification can be an effective tool in mitigating risk associated with break once, run anywhere attacks.

---

***"It seems that every time we introduce a new space in IT, we lose 10 years from our collective security knowledge. The Internet of Things is worse than just a new insecure space: it's a Frankenbeast of technology that links network, application, mobile and cloud technologies together into a single ecosystem, and it unfortunately seems to be taking on the worst security characteristics of each."***

***– Daniel Miessler***
*OWASP IoT Top 10 Project*

---

## 2.2 Anti-Tampering

The second major set of features of **Intertrust's whiteCryption Code Protection** include various ways to prevent hackers from tampering with your application. This technology is sometimes referred to as **Runtime Application Self-Protection (RASP)**.

### 2.2.1 Integrity Checking

**Intertrust's whiteCryption Code Protection** uses a *patented* technique for making sure code has not been tampered with. The basic premise is shown in Figure 2. Small pieces of code, called *checkers*, are inserted into your application. Each checker tests during runtime whether a small segment of the executable has been tampered with. If tampering has been detected, a number of possible actions can be taken including notifying a user, generating a log message, or immediately shutting down the program.

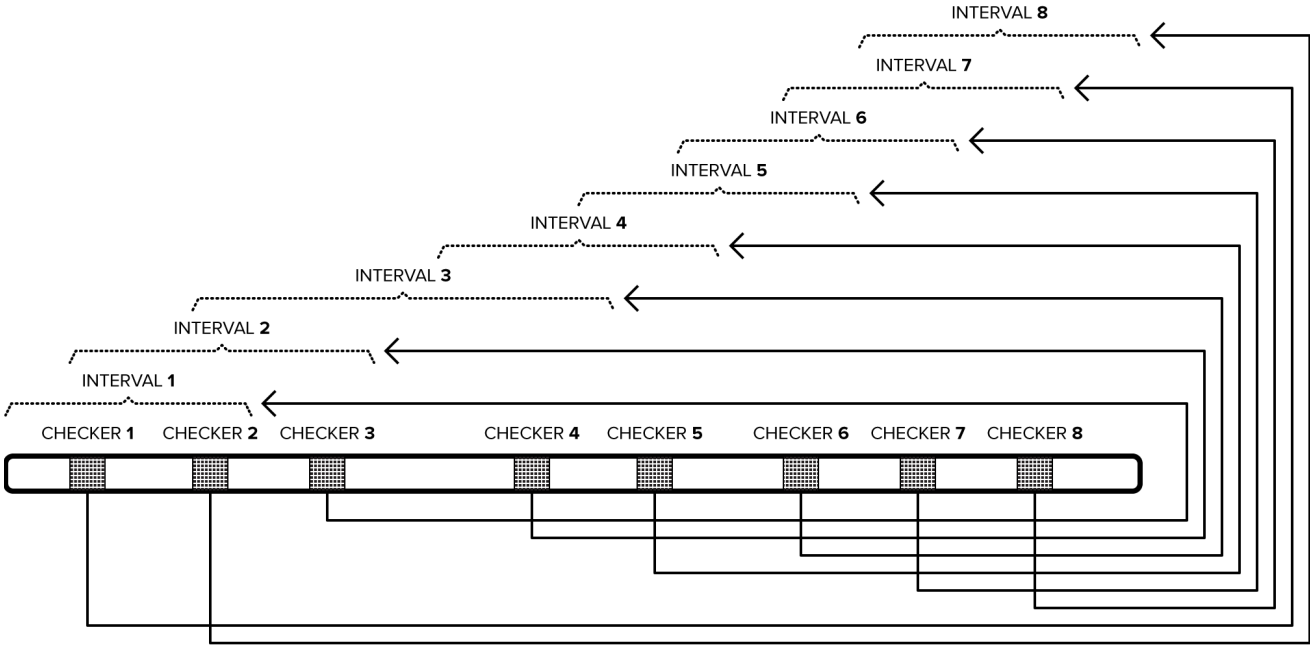



**Figure 2: Integrity protection using checksum checkers**

The scheme is illustrated in Figure 2. Here checker 1 is doing a checksum of interval 7. If a hacker attempts to modify checker 1, then checkers 3 and 4 will detect that a change has been made.

The executable is completely covered by thousands of overlapping checksum regions. This way, even if a hacker were to find one checker and disable it, there are several other checking parts of that region. And, several checkers checking that first checker, so now the hacker has to find those checkers and disable them. Those checkers, are in turn, checked by several other checkers. Each checker is diversified, so no two checkers look the same. And so effectively the entire set of checkers must be discovered and disabled at once in order to defeat the self-checking mechanism.

The checkers are inserted automatically without requiring any change to the code. Because of our profiling features (see section 3.3), checkers can be inserted to have minimal impact on performance.

Intertrust is a leader in this space publishing one of the first scientific papers on the topic[18], widely referenced in the scientific literature.

### 2.2.2 Anti-Method Swizzling

**Method swizzling** is a technique used in Objective-C applications, commonly used on Apple platforms, where the executable is modified so that certain method names are mapped to different method implementations. It is frequently used to replace or extend methods in binaries for which the source code is not available. This dynamic program modification is similar to **monkey patching**, a concept supported by other dynamic languages.

Although method swizzling is often used for legitimate purposes, it can also be used to attack an Objective-C application and modify its behavior in an undesirable way.

Code Protection provides a feature that allows the protected application to detect method swizzling and execute a defense action (See our manual for further detail, section 1.2.3.5)

### 2.2.3 iOS Jailbreak Detection

Jailbreaking is a process of gaining root access to an iOS device and overcoming its software and hardware limitations established by Apple. Jailbreaking permits a hacker to alter or replace system applications and settings, run specialized applications that require administrator permissions, and perform other operations that are otherwise inaccessible to a normal user.

Normally, a cracked or modified iOS application can be run only on jailbroken iOS devices. Code Protection provides security mechanisms that will execute the defense action (See our manual for further detail, section 1.2.3.4) if a jailbroken device is detected. The objective of the defense action is to prevent piracy and to help ensure that the application is run only on valid iOS devices.

### 2.2.4 Android Rooting Detection

Rooting is the process of allowing users of Android devices to attain privileged control of the operating system with the goal of overcoming limitations that carriers and hardware manufacturers put on the devices. Since users of rooted Android devices have almost complete control over the device and data it stores, a successful rooting of Android is a security risk to applications that deal with sensitive data or enforce certain usage restrictions.

Code Protection provides a specific anti-rooting feature that will execute the defense action (See our manual for further detail, section 1.2.3.6) if a rooted device is detected.

### 2.2.5 Function Caller Verification

An executable application file contains a number of functions. Normally, there is a predefined logic how and when these functions are called at run time. However, a skilled hacker can analyze the binary code, find vulnerabilities in the execution logic, and alter the original flow of the program by calling some functions in an unexpected way, for example on Windows, by using DLL injection.

**Code Protection** guards against such manipulation of function calls by creating a whitelist of modules (*.dll or *.exe files) that are allowed to call certain sensitive functions of the application code. Signatures of these authorized modules are stored within the application binary and used at run time to verify function caller modules.

### 2.2.6 Shared Library Cross-Checking

One way hackers can attack an application is by replacing or modifying the shared libraries it uses. Code Protection provides a feature called cross-checking of shared libraries that renders this type of attack significantly more difficult.

With shared library cross-checking enabled, you can select specific shared library files (*.dll or *.so) from your application, and **Code Protection** will calculate cryptographic signatures of their binary code and embed these signatures in the main application during the protection phase. Then, at arbitrary places in the application code, you can invoke a special function that checks if the signature of a particular shared library loaded in the memory matches the previously recorded signature. In other words, this function will check if the loaded shared library has not been modified or replaced.

### 2.2.7 Mach-O Binary Signature Verification

Every macOS, iOS, and tvOS application distributed via the App Store is signed with Apple's private key. This prevents piracy and unwarranted distribution of the apps. However, any user who is a member of the Developer Program, can re-sign any application with his own private key included in the development certificate, which allows the application to be run on corresponding development devices. Normally, this does not create a significant piracy risk, but there are several services on the Internet that employ re-signing of apps as a means for illegally distributing paid apps for free.

**Code Protection** provides a security feature that seeks to prevent unwarranted re-signing and distribution of apps in the Mach-O file format (used by macOS, iOS, and tvOS apps).

### 2.2.8 Google Play Licensing Protection

Application piracy is one of the primary concerns for Android developers. Although Android provides an anti-piracy library for verifying and enforcing licenses at run-time, this java library can be easily cracked and removed.

**Code Protection** provides a security feature specifically for addressing certain piracy vulnerabilities of Android apps. The security feature relies on an alternative implementation of the Google Play license verification library written in native code, which is difficult to reverse engineer and modify.

### 2.2.9 Customizable Defense Actions

By default, when **Code Protection** detects a threat, it corrupts the program state, which results in an application crash. Optionally, you can configure your protected application to execute a custom callback function defined in the source code and choose whether the program state should be corrupted or the application should be left running. For example, if you are protecting a game, in case of an attack, you may want to secretly corrupt some data (like the game map) instead of crashing the application. In this way, a hacker would think that the game is cracked whereas in reality it would be transformed into an unplayable state.

You can set a different callback function for each of the following threat types:

- code or data tampering
- debugger
- jailbroken iOS device
- rooted Android device
- method swizzling

# 3 Using whiteCryption Code Protection

## 3.1 The Protection Process

**Code Protection** protects an application by building a modified edition of the application from a copy of your source code where a series of security features are applied to both the source code and llvm bitcode (see section 1.2.3 of the User Manual) before they go through the compiler. The end result is an application that is functionally equivalent to the original, but at the same time hardened with various protection features against a wide range of attacks.

From the user's point of view, protecting an application with **Code Protection** involves executing the following three main steps:

1. Analyze the source code to ensure readability and availability (see section 3 in the manual).
2. Profile the application through run time analytics reported to **Code Protection** (see section 4 in the manual).
3. Build the protected version of the application (see section 5 in the manual).

***"Locking the door doesn't do any good if the key is under the doormat where anyone can find it."***

***– Tony DeLaGrange***

In general, you have to run through this process every time you modify your application's source code or general project settings. The only exception is profiling; if the amount of changes in the source code is insignificant and there are no new functions created, profiling does not have to be repeated every time.

## 3.2 Analysis

The purpose of the analysis step is to find out what files of the application's source code are compiled and how. **Code Protection** uses the obtained information later in the profiling and protection steps. Code Protection performs analysis by making a copy of the original source folder and building it using your provided build tools (the original source code is not modified). During the build process, Code Protection works as an intermediary between the source code and the build system, which allows Code Protection to get the necessary information.

## 3.3 Profiling

Profiling is a process that allows Code Protection to automatically analyze application's run-time behavior to detect its speed-sensitive functions. Profiling significantly improves the execution speed of the final protected application.

Profiling is based on the principle that functions that are most frequently called are typically not critical to an application's security, and therefore such functions can have a reduced protection level applied, which in turn results in faster execution speed. If your application contains frequently called functions for which you want stronger protection, you can use code marking (see section 1.2.7 in the manual) to specifically control the level of protection applied.

Profiling takes place as follows

1. Code Protection makes a copy of the original source folder and builds it to create a special version of the application, which contains additional profiling code embedded into the source code. The built application is called the profiling application.
2. A developer installs and runs the profiling application on a target device.
3. The profiling application performs one of the following procedures depending on how it is configured:
    - sends profiling statistics over the network to the workstation where Code Protection is running and capturing the analytics
    - saves the profiling statistics as a file on the target device

When profiling is completed, **Code Protection** uses the received statistics to automatically adjust the security level of different parts of the application.

## 3.4 Protection

Protection is the final step in the general protection process. In this step, Code Protection makes a copy of the original source folder, modifies it by applying and inserting the various security features selected by the user, and builds it. While building the protected version of the application, Code Protection takes into consideration the data obtained from the analysis and profiling steps (see section 3 in the manual) and adjusts the protection level for individual functions accordingly. The end result is a protected version of the application that is functionally equivalent to the original.

## 3.5 Code Marking

Although the profiling step will generally choose good locations for most of the security features, in some instances developers might want to have more control over where certain security features should or should not go.

Code marking is a feature that allows developers to insert specific #pragma statements, called code markers, directly into the source code to facilitate Code Protection profiling and make it more effective, as well as to explicitly tell Code Protection what function group or level of security should be applied to individual functions.

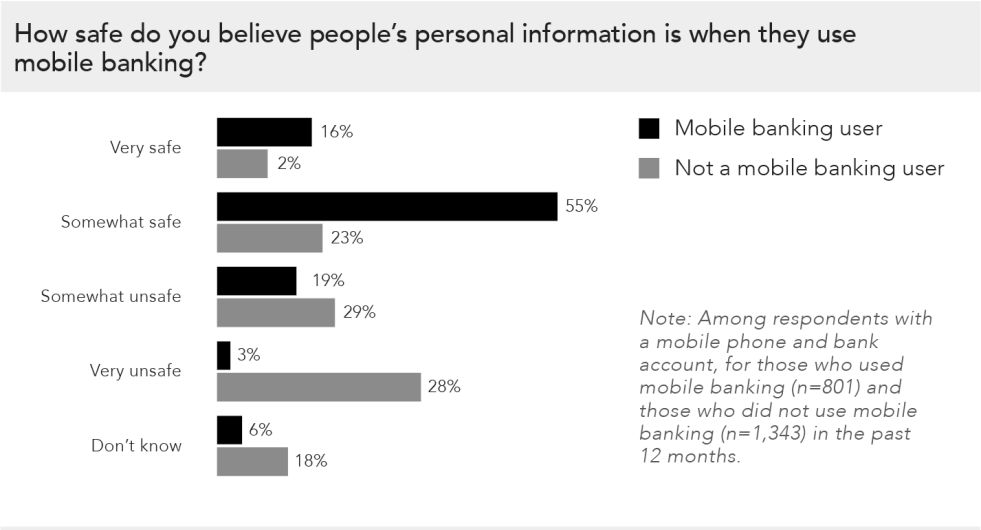
# 4 Select Use Cases

## 4.1 Mobile Payment Systems

Mobile banking is becoming the most important deciding factor when consumers switch banks, with 60% of research respondents citing this over fees (28%), branch location (21%) and services (21%).

The main defense against mobile banking malware starts with mobile app developers who need to adequately understand the risks that proliferate in the mobile data, connections and transactions ecosystem.

When an app is distributed to millions of devices and mobile banking users, it's not guaranteed that those devices are safe environments even when running security software. This is especially true with the trend toward jailbroken devices. By hardening the app with application shielding during the application development process, the app is **able to bring security with it no matter where it goes.** The solution is application shielding.




*Source: FederalReserve.gov*

## 4.2 Healthcare

The consequences of broken cryptography and unprotected applications are especially problematic in healthcare applications where doctors are moving toward mobile apps to address patient care issues. According to the Robert Wood Johnson Foundation, health apps will be on half of all mobile devices worldwide by 2018; however, personal health data requires higher standards for security and privacy because of the 1996 HIPPA requirements. The higher prevalence of broken cryptography suggests there will be unintended data leakage and other issues as the demand for mobile healthcare applications continues to rise.

Mobile device use is relatively new to the healthcare industry. In 2013, only 8% of doctors used mobile devices to manage patient data. By 2015, the number had grown to 70%, and now an estimated 90% of

healthcare providers are using mobile devices in their medical practice. Patients are also using their devices to make and confirm appointments and to access medical records through mobile apps. The U.S. Department of Health and Human Services reported more than 260 major healthcare breaches in 2015 with 9% of those breaches involving a mobile device other than a laptop. This number is expected to grow substantially in the future.

The majority of FDA-approved apps lack binary protection and have insufficient transport layer protection. Applications should have in-app security measures to protect against threats in the highly distributed mobile environment.

To protect patient data, it is essential to secure APIs that the mobile app uses to communicate with the server. Make sure to hide cryptographic keys within the application, and don't store the keys in memory, as this is a common path to back-end servers.

Develop an app that stores the most sensitive information server-side rather than in the mobile app to reduce liabilities, or if you must store secrets on the device, use whitebox cryptography to ensure their security.

## 4.3 Automotive

Today's car has the computing power of 20 personal computers and features 100 million lines of programming code. The connected car, controlled by software and high-tech features, may be one of the more significant advancements from the past few years. Features such as web browsing, Wi-Fi access points and remote-start mobile phone apps, help to enhance the enjoyment of the vehicle while *adding more opportunities for advanced attacks*.

In real life, we've seen thieves hack keyless entry systems in the UK to steal cars. Software recalls of cars have doubled within the past year and soon they will match mechanical recalls.

Stealing Personally Identifiable Information (PII): Connected cars collect a significant amount of data and interface with multiple after-market devices. Financial information, personal trip information and diagnostics can all be accessed through a vehicle's system. **Protect it!**

Manipulating a Vehicle's Operation: Catastrophic incidents resulting in personal injury and lawsuits may be in the near future. Famously, Charlie Miller and Chris Valasek demonstrated a proof of concept by hacking a Jeep. They now work for GM subsidiary Cruise ensuring customer confidence is strong for that effort.

Unauthorized Vehicle Entry: Car thieves now have a new way to gain entry into locked vehicles. Methods of obtaining entry include intercepting the wireless communication between the vehicle, or intercepting the fob signal from the driver. Many vehicle technologies have opted to replace physical ignition systems with keyless systems using mobile applications or wireless key fobs. In addition to gaining access to the vehicle, flaws in mobile apps have also led to controlling features independently, as discovered when Nissan **had to pull** its NissanConnect EV app for the Nissan Leaf in February 2016. The poor security of the app allowed security researchers to connect to the Leaf via the Internet and remotely turn on the car's heated seating, heated steering wheel, fans and air conditioning.

## 4.4 Media and Entertainment

Global entertainment and media companies have *increased their value* through innovative global streaming services, programs, live concerts, daily behind-the-scenes interviews, live sports broadcasts and a variety of music and news events that can be viewed on mobile devices. More importantly, consumers can now view specific entertainment content on their own devices just about anywhere, including planes, taxis, and other forms of public transportation.

To protect the content from being stolen, digital rights management (DRM) systems must be in place, and to protect the players' apps themselves, mobile security app solutions are a necessity. Developers should add a layer of protection to prevent hackers from reverse engineering and tampering with the service, content, or connected applications.

# 5 Related Products

In addition to Code Protection, Intertrust offers **whiteCryption® Secure Key Box™ (SKB)**, which is a cryptographic library that implements advanced **whitebox cryptography**. This allows standard cryptographic functions to be performed **without the keys ever being in the clear**. Like Code Protection, SKB is difficult to reverse engineer and tamper with. However, SKB provides significantly stronger protection for your keys and secrets, and is resistant to a wide variety of attacks, including sophisticated side channel attacks. Using both together makes the tight fist of protection that your code and business require.

Intertrust also offers **Seacert™**, a large scale key provisioning and managed PKI service. Seacert has delivered over three Billion keys to market in the last ten years. Our keys and credentials are embedded in hundreds of millions of set top boxes and IP enabled TV sets worldwide. We deliver keys in bulk to customers for factory floor provisioning, and we provide a highly scalable online provisioning service. We follow the industries most stringent standards for managing and delivering keys and are WebTrust audit certified.

See the Intertrust website for more information about these products and services, and look for forthcoming white papers like this one that goes into depth on each of them.

# 6 Next Steps

This white paper has presented an in depth look into our Code Protection product. Our state-of-the art protection mechanisms will help you shield your applications from attack and protect the most important assets for you and your customers.

Contact us to have a conversation about ways to solve these issues, get a demo, start a trial, or to learn more about all the areas we work in at Intertrust.

To find out more, go to https://www.intertrust.com/products/application-shielding, reach out on social media, or call us anytime.

## About Intertrust Technologies Corporation

Intertrust provides trusted computing products and services to leading global corporations – from mobile and CE manufacturers and service providers to enterprise software platform companies. These products include the world's leading digital rights management, software tamper resistance and privacy-driven data platforms for digital advertising, marketing technologies, DNA analysis, and IoT.

Founded in 1990, Intertrust is based in Silicon Valley, with regional offices in Beijing, Bengaluru, Hyderabad, Indore, Mumbai, London, Tokyo, Seoul, and Riga. The Company has a legacy of invention, and its fundamental contributions in the areas of computer security and digital trust are globally recognized. Intertrust holds hundreds of patents that are key to Internet security, trust, and privacy management components of operating systems, trusted mobile code and networked operating environments, web services, and cloud computing. Additional information is available at intertrust.com, or follow us on Twitter or LinkedIn.

- Founded in 1990; over 200 employees worldwide
- Headquartered in Silicon Valley with offices in Beijing, Bengaluru, Hyderabad, Indore, Mumbai, London, Tokyo, Seoul, and Riga
- Leader in security, privacy, and trust for open networks and service-enabled devices
- The inventor of Digital Rights Management (DRM)
- World-class research lab with largest patent portfolio in its field – with several top scientists including one Turing Prize winner
- Products in targeted advertising, healthcare, and IoT
- Investors include WiL, Sony, Philips, and innogy SE

## About whiteCryption

whiteCryption, a subsidiary of Intertrust Technologies, is a leading provider of application shielding solutions to prevent hackers from reverse engineering and tampering with code. We specialize in advanced obfuscation, runtime application self-protection (RASP), and white-box cryptography solutions for mobile apps, desktop applications, firmware and embedded applications. whiteCryption protects content for the automotive, banking/finance, healthcare, and entertainment industries.

## Copyright Information



## Contact Information

**whiteCryption Corporation**
920 Stewart Drive
Suite #100
Sunnyvale, California 94085, USA

www.whitecryption.com

## Sources


[1] B. Filkins, IT Security Spending Trends, SANS Institute, Feb 2016

[2] 2017 Cost of a Data Breach Study, Ponemon Institute, June 2017

[3] https://digitalhubshare.symantec.com/content/dam/Atlantis/campaigns-and-launches/FY17/Threat%20Protection/ISTR22_Main-FINAL-APR24.pdf?aid=elq_12438

[4] https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WGL03136USEN&

[5] http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide

[6] https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WGL03136USEN&

[7] https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html

[8] https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/medical-device-security-ponemon-synopsys.pdf

[9] https://www.csoonline.com/article/3119765/security/hackers-found-47-new-vulnerabilities-in-23-iot-devices-at-def-con.html

[10] https://www.wired.com/story/wind-turbine-hack/

[11] http://www.visualcapitalist.com/millions-lines-of-code/

[12] https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/

[13] https://nvd.nist.gov/vuln/search/results?adv_search=false&form_type=basic&results_type=overview&search_type=all

[14] https://www.mayerdan.com/ruby/2012/11/11/bugs-per-line-of-code-ratio

[15] https://www.extremetech.com/extreme/180435-the-nsa-knew-about-and-exploited-the-heartbleed-bug-for-at-least-two-years

[16] Web Applications Security Statistics Report, WhiteHat Security, 2016

[17] One only need to consider the four 2-bit numbers 00, 01, 10, and 11. The squares of those numbers are 00, 01, 100, and 1001 respectively. Notice that the second least significant bit in each case is zero.

[18] Horne, B.G., Matheson, L.R., Sheehan, C., & Tarjan, R.E. (2001). Dynamic Self-Checking Techniques for Improved Tamper Resistance. Digital Rights Management Workshop. https://dl.acm.org/citation.cfm?id=734764