whiteCryption®

Code Protection 2.18

User Guide

Copyright and Trademark Information

Disclaimer

Notice to U.S. Government End Users

Contact Information

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

contact@whitecryption.com

www.whitecryption.com

Table of Contents

1 Introduction

This chapter provides a general overview of Code Protection Licensed Technology ("Code Protection") and explains the basic concepts.

1.1 What Is Code Protection?

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 1.1).

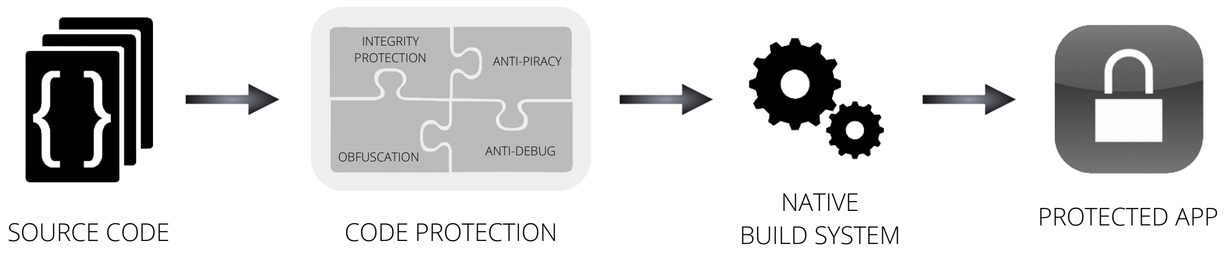


Figure 1.1: Code Protection overview

Code Protection can protect any standards compliant C/C++/Objective-C source code and requires no significant changes to the code itself or the existing build chain. Most of the security features are automated and can be executed via a tool with graphical user interface.

1.2 Basic Concepts

This section describes some of the main concepts that you should understand before working with Code Protection.

1.2.1 Nomenclature

This User Guide uses the terms "secure", "protect", and variations of each to convey very specific concepts — indeed, concepts that are far more specific and limited in their meanings than many meanings often associated with these terms in everyday usage. At the risk of stating the obvious, as used herein, none of these terms describe an absolute condition. Use of Code Protection in compliance with this User Guide will not render any application or data absolutely secure or absolutely protected from unauthorized accessing, use or manipulation. Use of these terms is not intended to convey a promise or warranty that Code Protection will never contain a bug or error, or that Code Protection will always operate without error.

As used in this User Guide:

- "secure" and variations of "secure" mean that the applicable application possesses characteristics that are intended to protect it against unintended use or access.
- "protect", "protected" and variations of these terms mean application of Code Protection features that

guard against understanding or modifying application code and function.

1.2.2 Protection Process

Code Protection protects an application by building a modified edition of the application where a series of security features are applied to both the source code and binary code (see §1.2.3). The end result is an application that is functionally equivalent to the original, but at the same time hardened with various protections against a wide range of hacker attacks.

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

1. Analyze the original source code (see §1.2.2.1).
2. Profile the application (see §1.2.2.2).
3. Build the final protected application (see §1.2.2.3).

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.

Since protecting an application can be a time-consuming process, this procedure should be performed only at the very end of your development cycle.

1.2.2.1 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 running a regular build of the source code using user's provided build tools. 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. For the technical aspects of this process, see §1.2.4.

1.2.2.2 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 §1.2.7) to specifically control the level of protection applied.

Profiling takes place as follows (see Figure 1.2):

1. Code Protection builds a special version of the target application containing specific 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 network to the workstation where Code Protection is running at run time
    - saves the profiling statistics as a file on the target device

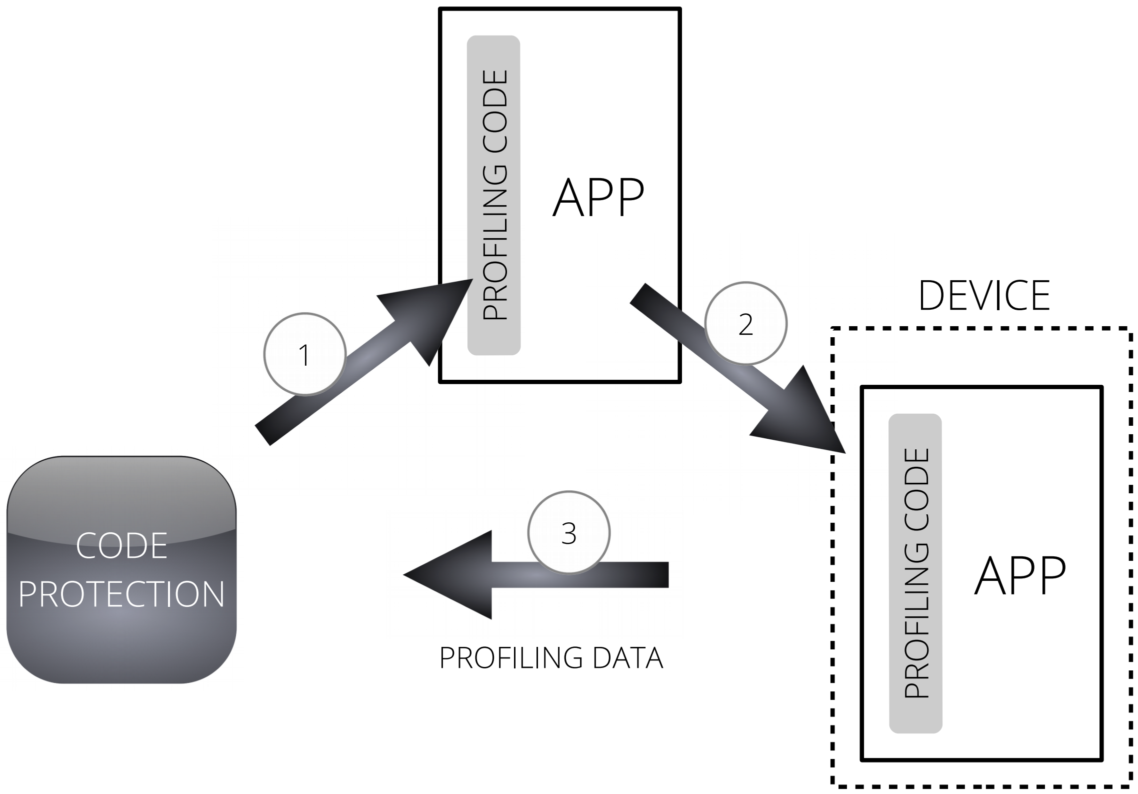


Figure 1.2: Profiling an application

When profiling is completed, Code Protection uses the received statistics to automatically adjust the security level of different parts of the application. For information on executing the profiling steps, see §4.

1.2.2.3 Protection

Protection is the final step in the general protection process. In this step, Code Protection creates and builds a copy of the original application's source code (based on the information obtained in the analysis step) and modifies this copy by applying and inserting the various security features selected by the user. While building the protected version of the application, Code Protection takes into consideration the data obtained from the profiling step (see §1.2.2.2) 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.

1.2.3 Security Features

This section describes every individual protection feature provided by Code Protection.

Table 1.1 provides an overview of available security features and supported target platforms for each feature.

Feature	**Windows**	**OS X**	**Linux**	**Android**	**iOS**	**Custom**
Integrity protection (§1.2.3.1)	✓	✓	✓	✓	✓	✓
Code obfuscation (§1.2.3.2)	✓	✓	✓	✓	✓	✓
Anti-debug protection (§1.2.3.3)	✓	✓	✓	✓	✓	✗
iOS jailbreak detection (§1.2.3.4)	N/A	N/A	N/A	N/A	✓	N/A
Method swizzling detection (§1.2.3.5)	N/A	✗	N/A	N/A	✓	N/A
Android rooting detection (§1.2.3.6)	N/A	N/A	N/A	✓	N/A	N/A
Mach-O binary signature verification (§1.2.3.7)	N/A	✓	N/A	N/A	✓	N/A
Google Play licensing protection (§1.2.3.8)	N/A	N/A	N/A	✓	N/A	N/A
Integrity protection of Android APK packages (§1.2.3.9)	N/A	N/A	N/A	✓	N/A	N/A
Verification of function caller modules (§1.2.3.10)	✓	✗	✗	✗	N/A	✗
Cross-checking of shared libraries (§1.2.3.11)	✓	✗	✗	✓	N/A	✗
Binary packing (§1.2.3.12)	✓	✗	✗	✓	N/A	✗
Inlining of static void functions (§1.2.3.13)	✓	✓	✓	✓	✓	✓
Objective-C message call obfuscation (§1.2.3.14)	N/A	✓	N/A	N/A	✓	N/A
Objective-C metadata encryption (§1.2.3.15)	N/A	✗	N/A	N/A	✓	N/A
String literal obfuscation (§1.2.3.16)	✓	✓	✓	✓	✓	✓
Customizable defense action (§1.2.3.17)	✓	✓	✓	✓	✓	✓
Diversification (§1.2.3.18)	✓	✓	✓	✓	✓	✓

Table 1.1: Supported security features

In the table, a green mark identifies a supported feature, a red mark identifies a feature that is currently not supported, and "N/A" means that the feature is not applicable for that particular platform.

1.2.3.1 Integrity Protection

Integrity protection is a set of measures that render application code and read-only data difficult to modify. Hackers usually need to modify the code or data, either in memory or on a storage medium, to circumvent its protection mechanisms. If a program detects that it has been modified, it can react to the attack as described in §1.2.3.17.

Code Protection achieves integrity protection by automatically inserting many pieces of code (called checkers) into the application code that calculate checksums of various overlapping portions of the binary. Two types of checkers are available:

- Code checker monitors the part of memory where the application's executable code section is loaded.
- Data checker monitors the part of memory where the application's read-only data is loaded, such as constant arrays and string literals.

The checksums are hard to detect and time-consuming to remove. If an application is modified, the embedded checksums will no longer match the binary footprint, and the application will execute the defense action (see §1.2.3.17).

1.2.3.2 Code Obfuscation

Code obfuscation is a process of transforming code so that it becomes more difficult to analyze by various analysis tools, which hackers use to identify critical functions in the executable.

Code Protection uses a number of code obfuscation techniques, including a patented control flow flattening algorithm. Code Protection obfuscation methods can be universally applied to any standards-compliant C/C++/Objective-C code.

1.2.3.3 Anti-Debug Protection

Anti-debug protection is a technique that allows a program to detect that it is being analyzed by a debugger and to perform counteractions.

Code Protection inserts numerous anti-debug checks into the protected application. These checks take into account the unique indications of the target platform that may identify the presence of a debugger. If a debugger is detected, the defense action is executed as described in §1.2.3.17.

1.2.3.4 iOS Jailbreak Detection

Jailbreaking is a process of gaining root access to the iOS device and overcoming its software and hardware limitations established by the manufacturer. 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 §1.2.3.17) 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.

1.2.3.5 Method Swizzling Detection

Method swizzling is a technique used in Objective-C applications 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.

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 the defense action (see §1.2.3.17).

1.2.3.6 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, successful rooting 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 §1.2.3.17) if a rooted device is detected.

1.2.3.7 Mach-O Binary Signature Verification

Every iOS and OS X 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 iOS or Mac 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 iOS and OS X apps).

For more information on using this feature, see §13.

1.2.3.8 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 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.

For more information on using this feature, see §11.

1.2.3.9 Integrity Protection of Android APK Packages

Android applications are delivered and installed as APK files, which contain the entire application code, resources, and digital signatures created by the developers. Hackers often modify Android APK packages and re-sign them with a different key so that they can be freely distributed to unlicensed users.

Code Protection provides a set of source code and run-time features that protect the APK package against tampering, including re-signing with a different key.

For more information on using this feature, see §12.

1.2.3.10 Verification of Function Caller Modules

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, by using DLL injection.

Code Protection can guard 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.

For more information on using this feature, see §9.

1.2.3.11 Cross-Checking of Shared Libraries

One way how 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.

For more information on this feature, see §10.

1.2.3.12 Binary Packing

For enhanced protection against static analysis, Code Protection provides a security feature that allows you to pack binary executables for Windows and Android applications. Packing means that the binary code is encrypted, and an appended stub decrypts it into memory only at run time.

Currently, binary packing is limited to the following cases:

- For Windows applications, this feature encrypts the code section of all executable binaries.
- For Android applications, this feature encrypts the code section and the read-only data section of `.so` libraries, which are written in the native C/C++ code and later invoked from Java code.

⚠ Currently, Code Protection does not support binary packing on all architectures. For information on which architectures are supported, see §1.5.

1.2.3.13 Inlining of Static Void Functions

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 an increased security of the protected application.

1.2.3.14 Objective-C Message Call Obfuscation

Objective-C is designed so that messages to object instances are resolved at run time. Because of this fact, message calls are stored in plain form in the binary code. This is a security loop that hackers can use to manipulate with the execution logic. Code Protection provides a simple security feature that obfuscates message calls in the binary code, thus making reverse engineering more difficult.

1.2.3.15 Objective-C Metadata Encryption

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.

Currently, Code Protection only encrypts Objective-C method names and arguments, but does not encrypt class names.

Objective-C metadata encryption is not supported for protected static libraries.

1.2.3.16 String Literal Obfuscation

Usually, application source code contains a large number of string literals, which appear as string constants in the binary code. This information may reveal useful information (such as log or user messages) to potential attackers and provide a means for manipulating with the execution logic.

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.

Code Protection does not obfuscate string literals that have the section attribute, such as the following:

```
static const char var[] __attribute__((__section__(".rodata.slog1"))) = "string";
```

You should not rely on string literal obfuscation to protect sensitive information, like cryptographic keys. If you want to protect cryptographic keys, we recommend using Cryptanium Secure Key Box. For more information on this product, see the following web page: www.whitecryption.com/secure-key-box/

1.2.3.17 Customizable Defense Action

By default, when Code Protection detects a threat, it corrupts the program state, which results in 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
- method swizzling
- rooted Android device

⚠ Program state corruption will not be executed if debug logging is enabled in the protected application (see §5.3.1).

1.2.3.18 Diversification

Diversification is a method of altering an executable binary so that various instances of the same software, while providing identical functionality, appear different on the binary level. Diversification confounds an attacker's attempts to exploit information gained from breaking one deployment to compromise other deployments.

When Code Protection builds a protected version of the target application, most security features (such as obfuscation, integrity checkers and so on) within that application are generated from a random seed. This means that every time you build the protected application, its binary footprint is different. You can employ this feature to implement a proactive defense system where you regularly replace the protected application with new diversified instances. This will frustrate hacker attempts to crack your application.

1.2.4 How Code Protection Interacts With the Build System

Code Protection is designed to employ system's native build tools, which enables simpler integration with your build system. Figure 1.3 shows how Code Protection interacts with the native build system to produce a protected application.

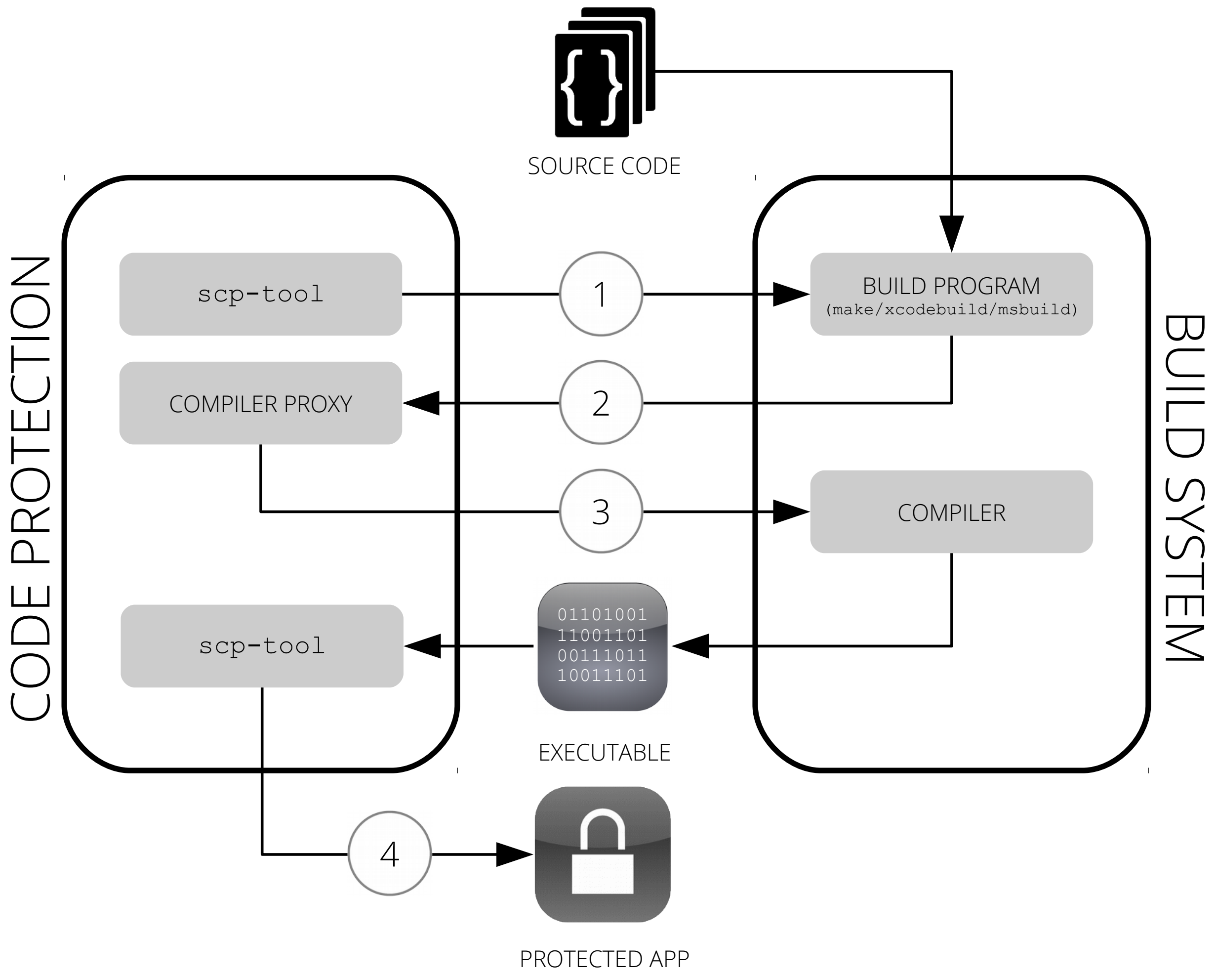


Figure 1.3: Code Protection integration with the native build system

The following procedure explains the steps highlighted in Figure 1.3:

1. `scp-tool` (the main component of Code Protection) invokes the system's build program, such as `msbuild` on Windows, `make` on Linux, or `xcodebuild` on OS X.

2. Instead of directly invoking the standard system's compiler, Code Protection instructs the build system to invoke a specific Code Protection component named the compiler proxy, which mimics the compiler executable.

   The compiler proxy works as a mediator between Code Protection and the actual compiler, allowing Code Protection to intercept and control the compiling process.

   On Windows, there is an additional proxy for the linker as well. This is not reflected in Figure 1.3.

3. The compiler proxy parses command-line arguments, analyzes the source code (in the analysis stage), and invokes the original compiler to continue the build.

4. In the protection stage, the compiled binary executable is again passed to `scp-tool` to update the embedded checksums for integrity protection (see §1.2.3.1).

When you use Code Protection to protect applications for Windows, OS X, iOS, or Android with default build settings (see §3.2.1), Code Protection configures the build components automatically. However, in the following circumstances you must perform certain manual steps:

- If you are protecting a Linux application or if you are using the custom build option (see §3.2.2), you must manually configure your build system to invoke the compiler proxy as described in §14.
- If you are protecting an application for a custom platform (that is not directly supported by Code Protection), you have to write your own plugin that does most of the actions included in step 3 above. For information on protecting applications for custom platforms, see §15.

1.2.5 Graphical User Interface

Code Protection has a graphical user interface (GUI), which contains the controls for setting up and applying security features to the target application. Using the GUI, you specify where your application's source code is located, what build system should be used, which security features need to be applied, and where the final protected application should be placed. For information on using the GUI, see §1.5.

For automated build tasks and scripting, you can use the Command-Line Tool as an alternative to the GUI (see §8).

1.2.6 Project

To build a protected application, you have to create and execute a Code Protection project. This project is a file with the extension `.nwproj` containing all configuration settings required to apply code protection to a particular application. Each project is associated with one application and one target platform.

Projects are created and modified only with the GUI. When protecting an application using Code Protection, the project file must always be supplied — either to the GUI or to the Command-Line Tool.

1.2.7 Code Marking

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 level of security should be applied to individual functions. For detailed information on code marking, see §7.

1.2.8 "__SCP__" Macro

Several Code Protection features (such as code marking, function caller verification, Mach-O binary signature verification, and others) require developers to make certain changes in the source code of the target application. Such changes may involve adding Code Protection specific `#pragma` statements, header files, or function calls. These changes may cause warnings or errors when building the application directly, outside of the Code Protection environment. For this purpose, we strongly recommend surrounding all Code Protection specific code with conditional directives that depend on the existence of a specific macro named "`__SCP__`". Code Protection defines this macro whenever it builds an application. Therefore, the existence of this macro is an indication of whether the Code Protection infrastructure is present during the build process.

In general, the "`__SCP__`" macro should be used as follows:

```
#ifdef __SCP__
  «Code Protection specific code»
#else
  «fall-back code (if necessary)»
#endif
```

Further in this document, we provide specific recommendations regarding the use of the "`__SCP__`" macro for every individual Code Protection feature that depends on source code modifications.

1.3 Supported Platforms

This section describes the development and target platforms supported by Code Protection.

1.3.1 Development Platforms

This section lists platforms on which you can build and protect applications using Code Protection.

Windows 7 or later

Can be used to protect applications for Windows and Android.

OS X 10.7 or later

Can be used to protect applications for OS X, iOS, and Android.

⚠ If you are building an application for OS X or iOS, Code Protection requires Xcode Command Line Tools to be installed. You can install Xcode Command Line Tools in the **Components** tab of the **Downloads** page in the Xcode preferences panel. Additionally, make sure the "`xcode-select -print-path`" command returns a correct path to the Xcode installation. If necessary, you can fix it using the "`xcode-select -switch`" command.

Linux

Can be used to protect applications for Linux and custom platforms (see §15).

The x86-64 architecture Linux development platform can also be used to protect Android applications.

The following are the minimal system requirements for Linux development systems:

- x86 or x86-64 architecture
- glibc 2.11.1 or later
- zlib 1.2.3 or later

⚠ Certain manual steps need to be performed to prepare your Linux build system for Code Protection as described in §14.

1.3.2 Target Platforms

This section lists the supported build systems and architectures for every target platform. A target platform is a platform on which applications protected by Code Protection will be run.

Windows

Visual Studio 2010, 2012, and 2013 (x86 and x86-64).

OS X

Xcode versions 6.3 to 7.3.

Linux

Autotools 1.11.3 or later, CMake 2.8.8 or later, and SCons 2.1.0 or later with GNU Compiler Collection (GCC) 4.8 or later (x86 and x86-64).

Android

Android NDK r9c or later (armeabi, armeabi-v7a, arm64-v8a, x86, x86_64, and mips).

iOS

Xcode versions 6.3 to 7.3 (ARMv7, ARMv7s, and ARMv8/arm64).

Custom

This is a special case that allows you to use Code Protection for target platforms that are not directly supported. For instance, you may have some proprietary build system that you do not want to expose to other parties.

Selecting a custom platform means that you will have to implement a few functions specific to your environment. Code Protection will then use these functions via a plugin interface when building, profiling, and protecting your application.

For details on protecting applications on custom platforms, see §15.

1.4 Examples

The Code Protection package contains example projects that demonstrate several of the more advanced Code Protection security features. These example projects are archived into a single file named `examples.zip`. Each example project is placed in a separate subfolder. In each subfolder, there is a file named `Instructions.txt` that explains how to build and run the specific example project.

1.5 Limitations and Known Problems

Please carefully review the following list of Code Protection limitations and known problems before building protected applications:

- Code Protection does not protect functions marked as `inline`. Furthermore, methods whose bodies are defined in class declarations are also not protected, since such methods are treated as `inline` functions.
- Code Protection cannot protect source files that are defined by absolute paths in their build projects and scripts. The reason for this is that Code Protection makes copies of the source files when building the profiling application and the protected application, but it does not update the paths in build projects and files.

- If you use CMake to generate project files for your application, Code Protection may not correctly build the application using these project files as they may contain absolute paths. The workaround for this problem is to use the custom build option in Code Protection (see §3.2.2) and invoke a script that executes the following steps:

  1. Invoke CMake to generate project files.
  2. Build the application with the project files generated in step 1.

  This workaround will work only if the CMake configuration does not use absolute paths for the source code files.

- A protected application that is built for the 64-bit Windows target platform does not correctly detect the caller module (see §1.2.3.10) if the call of the protected function is the last statement in the caller function (i.e. there are no other statements in the caller function after calling the protected function). This happens because of the way the compiler optimizes code.

- Binary packing is currently supported only on the following operating systems and architectures:
  - x86 architecture for Windows
  - x86 and ARM architectures for Android

- Currently, Code Protection only encrypts Objective-C method names and arguments, but does not encrypt class names.

- Objective-C metadata encryption is not supported for protected static libraries.

- Code Protection cannot protect Android applications if the `NDK_CCACHE` environment variable is defined.

- Code Protection does not support GCC 4.6 for Android applications due to various problems in the compiler. You should use other GCC versions or Clang instead.

- Code Protection supports the iOS bitcode build type only if integrity protection is disabled (see §1.2.3.1). If integrity protection is required, you must make sure the **Enable Bitcode** option is set to **No** in the Xcode build options.

- Code Protection does not support the iOS Simulator.

- If you are building an application for OS X or iOS and you have Clang modules enabled, you must disable the **Link Frameworks Automatically** option in Xcode.

- If you are using the custom build option for OS X or iOS, you must set the Xcode setting **Derived Data** to **Relative**. To view and modify this setting, open Xcode and select **Preferences > Locations**.

- If you are building an application for OS X or iOS, you must either disable precompilation of prefix headers or add the "`GCC_PRECOMPILE_PREFIX_HEADER=NO`" parameter to the `xcodebuild` command.

- If you are building an application for OS X or iOS and you have recursive header search paths enabled, you must make sure the include folder structure is not changed after the analysis step. For example, the build folder must not be a subfolder in the header search paths.

2 Using the GUI

This chapter describes the operations you can perform with the GUI, which is the primary Code Protection part for configuring security features and applying protection to applications.

2.1 Opening the GUI

To open the GUI, perform one of the following steps depending on the operating system you are using:

- On Windows, double-click the **Cryptanium Code Protection** desktop icon or launch the `scp.exe` file, which is located in the folder where you installed Code Protection.
- On OS X, launch the **Cryptanium Code Protection** application.
- On Linux, launch the `scp` file.

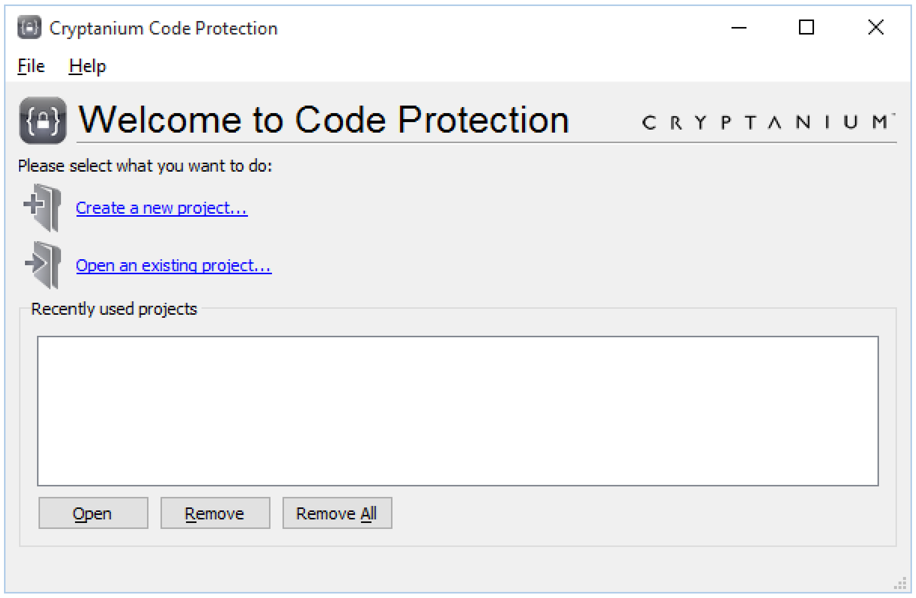
The welcome page is displayed. For information on using the welcome page, see §2.2.1.

2.2 GUI Overview

The GUI displays a number of controls that enable you to create and use Code Protection projects, configure project settings, and execute various operations on the application to be protected. The subsequent sections describe how the GUI is organized.

2.2.1 Welcome Page

Welcome page (see Figure 2.1) is the first page displayed when you launch Code Protection. Its primary purpose is to provide controls for managing Code Protection project files (see §1.2.6).



Welcome to Code Protection

Please select what you want to do:

Figure 2.1: Welcome page

The screenshots used in this document are captured on the Windows operating system. The user interface on other platforms is very similar.

In particular, you can perform the following operations in the welcome page:

- create a new project as described in §2.3.1
- open an existing project as described in §2.3.2
- view the list of recently used projects

  You can quickly open any project in the **Recently used projects** section by double-clicking it or by selecting it and clicking **Open**.

  If a recently used project no longer exists, you can remove it from the list by selecting it and clicking **Remove**, or right-clicking it and selecting **Remove Project**. You can also clear the entire list of recently used projects by clicking **Remove All** or right-clicking the list and selecting **Remove All**.

2.2.2 Project Pages

When you have opened a particular Code Protection project, the project settings are organized into logical pages described below.

Analysis

Allows you to provide the basic information about the target application (source code location, build options, and features to be enabled) as well as provides controls for analyzing the source code by building it.

For information on using this page, see §3.

Profiling

Specifies profiling information and allows Code Protection to perform analysis of the target application's run-time behavior.

For information on using this page, see §4.

Protection

Specifies security settings to be applied and allows you to build the final protected application.

For information on using this page, see §5.

In general, the pages are listed in the order they are typically intended to be used, but you can switch between the pages any time as follows:

- Click the page titles in the menu on the left side (Figure 2.2.)

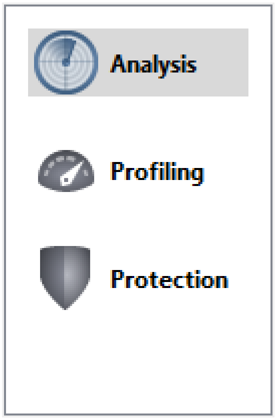


Figure 2.2: Left-hand side menu

- Use the **Project** submenu in the application menu (Figure 2.3).

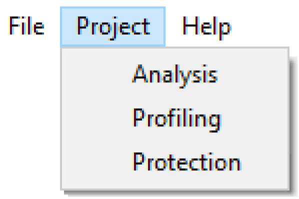
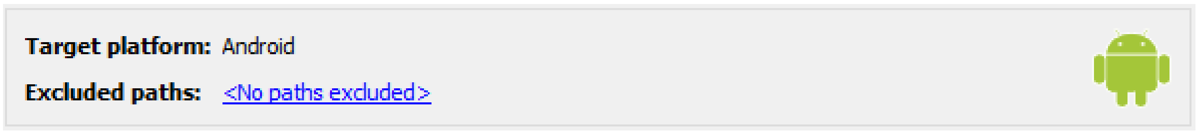


Figure 2.3: Project submenu

You cannot navigate away from the **Analysis** page if the information entered is invalid. In that case, Code Protection displays an error message and highlights the incorrect field in red.

2.2.3 Information Pane

Every project page contains an information pane, which is a separate area in the upper part of the window (Figure 2.4).



Target platform: Android

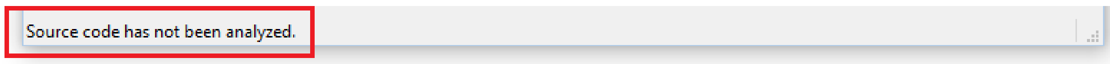
Excluded paths: <No paths excluded>

Figure 2.4: Information pane

The information pane displays the selected target platform (also represented by the icon on the right side) and provides a control for excluding specific folders or files from processing (see §2.4).

2.2.4 Status Bar

Status bar is the area in the lower part of the window containing text (Figure 2.5).



Source code has not been analyzed.

Figure 2.5: Status bar

The status bar displays important messages about the project. Mostly, the messages are related to profiling, for example, the date and time when the application was last profiled, or any problems that have occurred with profiling. The messages displayed may also include errors regarding project databases. The information in the status bar allows you to quickly find out what the current project status is and what needs to be done.

2.2.5 Progress Bar

All project pages will display a progress bar in the lower part of the window during building and profiling processes.

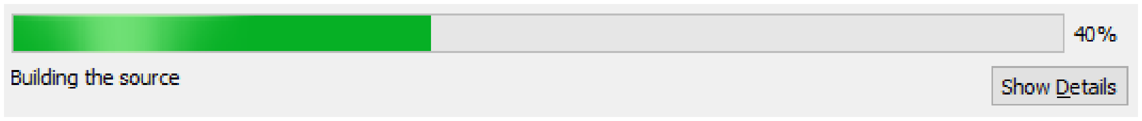


Figure 2.6: Progress bar

The message below the bar identifies the current stage of the process started. If you want to see the full output of the process, you can click **Show Details**, which will show full details below the progress bar.

When a process is completed, one of the following messages is displayed below the progress bar:

SUCCESS

The process was successfully completed without problems.

FAILURE

The process failed. Click **Show Details** and examine the output to see what caused the failure.

N erroneous file(s)

N source code files caused build errors. Click this message to display a separate window showing only the errors. Alternatively, you can click **Show Details** to examine the full build output.

2.3 Working with Projects

This section describes tasks that you can perform with projects in the GUI.

2.3.1 Creating a New Project

To apply protection to an application, you have to create and execute a Code Protection project (see §1.2.6).

⚠ If you are protecting a static library, you must follow special instructions described in §6.

To create a new project, proceed as follows:

1. Open the GUI as described in §2.1.
2. In the welcome page, click **Create a new project** or select **File > New Project** in the application menu.

   The **Target Platform** window is displayed allowing you to select the target platform on which the protected application will be run. For information on supported target platforms and build systems, see §1.3.2.

   ⚠ If you are protecting a Linux application, make sure you follow the instructions in §14 before you start the protection process.

3. Select the required platform and click **OK**.

   A pop-up window is displayed asking you to save the new project.

4. In the pop-up window, go to the folder where you want to save the new project, give the project an appropriate name, and click **Save**.

   ⚠ Do not save the project inside the application source folder as it may interfere with the protection process.

   After the new project is saved, Code Protection takes you to the **Analysis** page (see §3).

2.3.2 Opening an Existing Project

You can open an existing project using any of the following approaches:

- Double-click the `.nwproj` file in your file system.

  On Windows and OS X, the project file type association is set automatically. On Linux, you have to manually associate `.nwproj` files with the `scp` executable.

- Pass the project file as an argument to the `scp` executable from the command line.
- Open the GUI as described in §2.1, and in the welcome page, perform one of the following steps as appropriate:
  - Click **Open an existing project** and browse to the project file.
  - In the application menu, select **File > Open Project** and browse to the project file.
  - In the application menu, select **File > Open Recent** and select the appropriate project file.
  - Double-click a project in the **Recently used projects** list, or select a project in the list and click **Open**.

Once the selected project is opened, Code Protection takes you to the **Analysis** page (see §3). The file name of the currently opened project is displayed in the window title bar.

2.3.3 Closing a Project

You can close an opened project any time in the GUI. If there are any unsaved modifications in project settings, Code Protection will offer you to save the changes.

To close a project, perform any of the following actions:

- In the application menu, select **File > Close Project**.

  After closing the project, you will be returned to the welcome page.

- In the application menu, select **File > Exit**, or close the main window.

  After closing the project, the GUI will be closed as well.

2.3.4 Upgrading a Project

The way Code Protection stores parameters in a project file differs between Code Protection versions. Therefore, projects created with older Code Protection versions may need to be upgraded to the current version before execution.

Code Protection will automatically offer you to upgrade an outdated project when you open it in the GUI.

Alternatively, you can pass the outdated project to the Command-Line Tool with the parameter "`--upgrade`" as described in §8.2.

Code Protection will always create a backup copy of the original Code Protection project before upgrading. The copy will have the extension `.backup`.

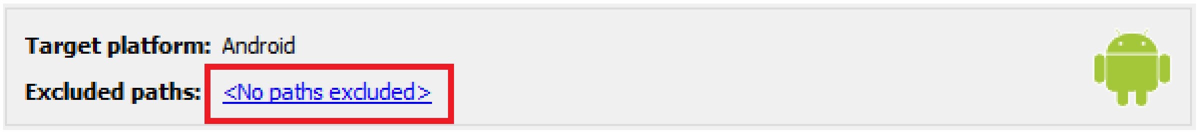
2.4 Excluding Specific Source Files from Processing

In rare situations, Code Protection may cause the build system to crash on particular source files. The GUI provides a mechanism for excluding such files from processing. This means that Code Protection will not modify these files at all and will not include any protection features in them.

⚠ You should avoid excluding files if it is possible, because unprocessed files will essentially be unprotected. The only security feature that will protect these files is integrity protection of the entire application.

To exclude certain files or folders from processing, proceed as follows:

1. Open the Code Protection project in the GUI.
2. In the information pane, click the link next to the **Excluded paths** field (see Figure 2.7).



Target platform: Android

Excluded paths: <No paths excluded>

Figure 2.7: Accessing the "Excluded Paths" window

   The **Excluded Paths** window appears showing all source files in the source root folder (see Figure 2.8).

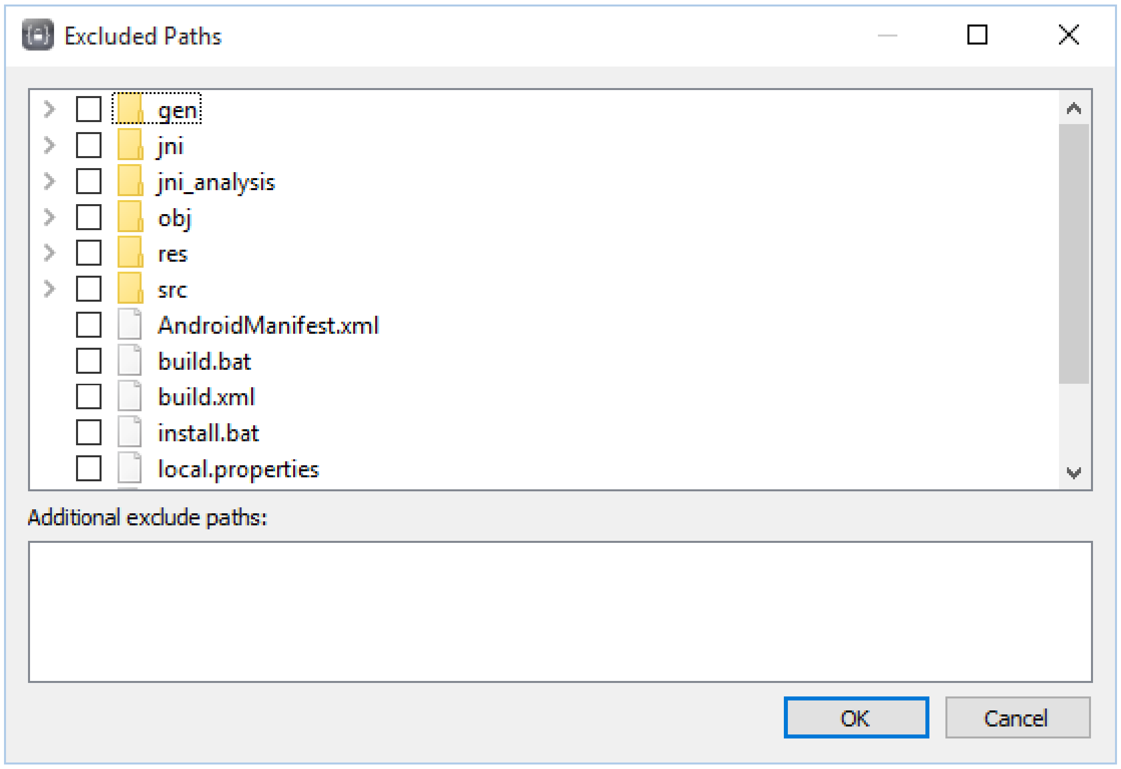


Figure 2.8: Selecting files and folders to be excluded from Code Protection processing

A selected check box next to a folder or file means that Code Protection will exclude it from processing and will not modify its contents.

3. To exclude a file or folder from processing, select the check box next to it.
4. To exclude files and folders that are not available for selection (such as items that are created during the build process), in the **Additional exclude paths** field, enter paths to these elements.

   In most cases, paths to be excluded are entered relative to the source root. As a special case, to exclude external header files, you may also enter absolute paths. Every path must be placed on a separate line, for example as follows:

   ```
   Generated
   Test/test1.cpp
   Test/test2.cpp
   ```

5. Click **OK**.

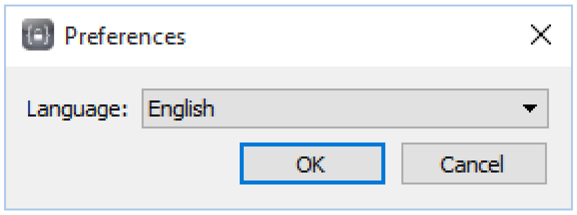
   The excluded files and folders are now displayed in the **Excluded paths** field.

2.5 Changing the UI Language

To change the language in which the UI is displayed, proceed as follows:

1. In the application menu, select **File > Preferences** (on OS X, **Cryptanium Code Protection > Preferences**).

The **Preferences** window appears (see Figure 2.9).



Language: English

Figure 2.9: Preferences window

2. In the **Language** list box, select the required language.
3. Click **OK**.

2.6 Finding Out the Code Protection Version Number

To find out what is the version and revision number of the Code Protection build that you are using, in the application menu, select **Help > About Code Protection**.

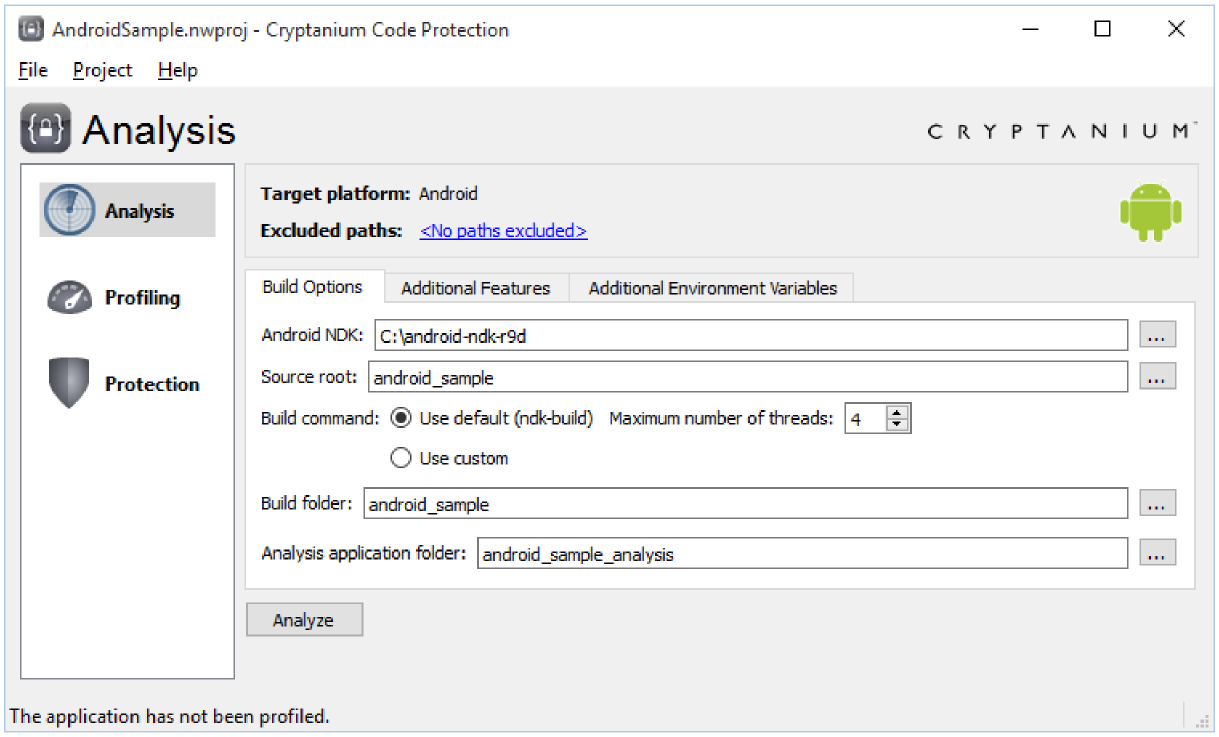
3 Analyzing the Application

This chapter describes how to use the GUI to execute the analysis step, which is the first step in the overall protection process (see §1.2.2).

If your Code Protection project does not require any modifications, you can also execute the analysis step using the Command-Line Tool as described in §8.

3.1 Overview of the Analysis Page

All analysis-related settings and controls are available in the **Analysis** page of the GUI (see Figure 3.1). It is the first page that is displayed when you create or open a project.



Target platform: Android
Excluded paths: <No paths excluded>

Android NDK: C:\android-ndk-r9d
Source root: android_sample
Build command: [✓ Use default (ndk-build)] Use custom
Maximum number of threads: 4
Build folder: android_sample
Analysis application folder: android_sample_analysis

The application has not been profiled.

Figure 3.1: Analysis page

This page serves the following primary purposes:

- It provides basic information about the target application, such as the following:
  - location of the target application's source code
  - build system to be used to build the application
  - project-wide features to be enabled

    This information is essential for all other operations that can be executed with Code Protection.
- It allows you to analyze the original application to understand how the source code is organized and which files need to be protected.

3.2 Executing the Analysis Step

To correctly execute the analysis step in the **Analysis** page, proceed as follows:

1. In the **Build Options** tab, perform the following steps:

   - If you are protecting an Android application, in the **Android NDK** field specify the folder where Android NDK is installed.

     Code Protection requires the NDK to build the native code.

   - In the **Source root** field, specify the root folder of the application's source code.

     ⚠ All source files to be protected must be located under this folder. Files outside this folder will not be processed by Code Protection.

   - Select one of the following build options:

     Use default

     Tells Code Protection to use the default build system of the target application. The build steps performed by Code Protection and the GUI controls available depend on the target platform.

     If you select this option, you can also adjust the value in the **Maximum number of threads** field. This value specifies the maximum number of threads to be used for compilation. The value is limited by the number of processor cores on your workstation.

     For more information on this option, see §3.2.1.

     Use custom

     Tells Code Protection to use a custom build command.

     For more information on this option, see §3.2.2.

2. If necessary, in the **Analysis application folder** field, modify the name of the folder where the source code will be copied to and analyzed.

   During the analysis step, Code Protection will copy the original source code to this folder and build it. Source code in the original location will be left intact.

   This field is automatically set when you select the **Source root** field.

3. In the **Additional Features** tab, enable and configure the following project-wide features as necessary:

   Advanced anti-debug (Windows)

   Enables enhanced anti-debug functionality in the protected application. During runtime, the application creates a new process, and tries to attach it as a debugger of the original parent process. Advanced anti-debug works in parallel to the standard anti-debug functionality.

   ⚠ Although this feature provides stronger anti-debug protection, it increases run-time overhead due to frequent process creation and does not work for protecting static libraries.

   Verification of function caller modules (Windows)

   Allows you to protect certain sensitive functions in your application against unauthorized caller modules as described in §9.

   Cross-checking of shared libraries (Windows, Android)

   Enables cross-checking of shared libraries as described in §10.

Google Play licensing protection (Android)

Enables the secure implementation of the License Verification Library to harden the Google Play license checking mechanism. For more information on this feature, see §11.

APK integrity protection (Android)

Enables Android application package integrity protection as described in §12.

Mach-O binary signature verification (OS X, iOS)

Enables protection against unauthorized re-signing as described in §13.

Encrypt Objective-C metadata (iOS)

Encrypts Objective-C metadata in the protected executable as described in §1.2.3.15.

Incremental analysis

If this check box is selected, you do not have to repeat the full source code analysis step every time you change the source code. Instead, when building the profiling application or the final protected application, Code Protection detects modified source code files and automatically rebuilds only them.

The full source code analysis step has to be repeated however in the following situations:

- when you modify the project-wide options in the **Analysis** page
- when you add new `#include` directives to your source code that require compiling additional files

⚠ We strongly recommend to always keep this option enabled. Otherwise, you will have to repeat the full source code analysis step every time you make any kind of changes to the source code.

4. Optionally, in the **Additional Environment Variables** tab, set additional environment variables to be used by the build system.

   Every variable must start on a new line and have the following pattern:

   ```
   «variable name»=«value»
   ```

   Now Code Protection has all the necessary information to analyze the source code.

5. Click **Analyze**.

   Code Protection invokes the build system and starts building the source code in the folder specified in the **Analysis application folder** field. A progress bar appears in the lower part of the page indicating the current build progress. For information on the progress bar, see §2.2.5.

3.2.1 Using the Default Build Command

Selecting the **Use default** option in the **Build Options** tab tells Code Protection to use the default build system of the target application. The build steps performed and the GUI controls available depend on the target platform as described in the following subsections.

3.2.1.1 Windows

If you are protecting a Windows application, Code Protection will call Visual Studio using the `msbuild` command. You have to provide additional information in the following fields:

Visual Studio solution

Path to the Visual Studio solution file.

> ⚠ This file must be located under the source root folder path.

Build configuration

Visual Studio build configuration to be used.

Visual Studio version

Visual Studio version to be used for building the solution.

3.2.1.2 Linux

If you are protecting a Linux application, Code Protection will call GCC using the `make` command.

In the **Build folder** field, you have to specify the folder from which the build process will be executed. This is the folder that contains the `Makefile` file.

> ⚠ This location must be either equal to the source root folder path, or be located under it.

3.2.1.3 Android

If you are protecting an Android application, Code Protection will call Android NDK using the `ndk-build` command.

In the **Build folder** field, you have to specify the folder from which the build process will be executed. This is the folder that contains the `AndroidManifest.xml` file or the `jni` folder.

> ⚠ This location must be either equal to the source root folder path, or be located under it.

3.2.1.4 OS X and iOS

If you are protecting an OS X or iOS application, Code Protection will call Xcode using the `xcodebuild` command. You have to provide additional information in the following fields:

Xcode project/workspace

Path to your application's Xcode project or workspace.

> ⚠ The selected file must be located under the source root folder path.

Build target

Xcode project build target to be used. This field is displayed if you have selected an Xcode project.

Build scheme

Xcode scheme to be used. This field is displayed if you have selected an Xcode workspace.

⚠ Only shared schemes are available.

3.2.2 Using the Custom Build Command

Selecting the **Use custom** option in the **Build Options** tab tells Code Protection that you want to specify a custom build command to build your application.

If you select this option, the **Custom build** command field appears below it. In this field, you must enter the build command to be invoked.

You must take into consideration the following warnings when using the custom build command:

- Your build system may need to be configured to support Code Protection as described in §14.
- You must always include the build system's rebuild option, so that the entire code is compiled each time.
- If symlinks are created while your application is being built, Code Protection may fail in profiling and protection stages. To resolve this problem, you must delete these symlinks from the source folder before the profiling and protection stages are started.

4 Profiling the Application

This chapter describes how to use the GUI to execute the profiling step, which is one of the main steps in the overall protection process (see §1.2.2).

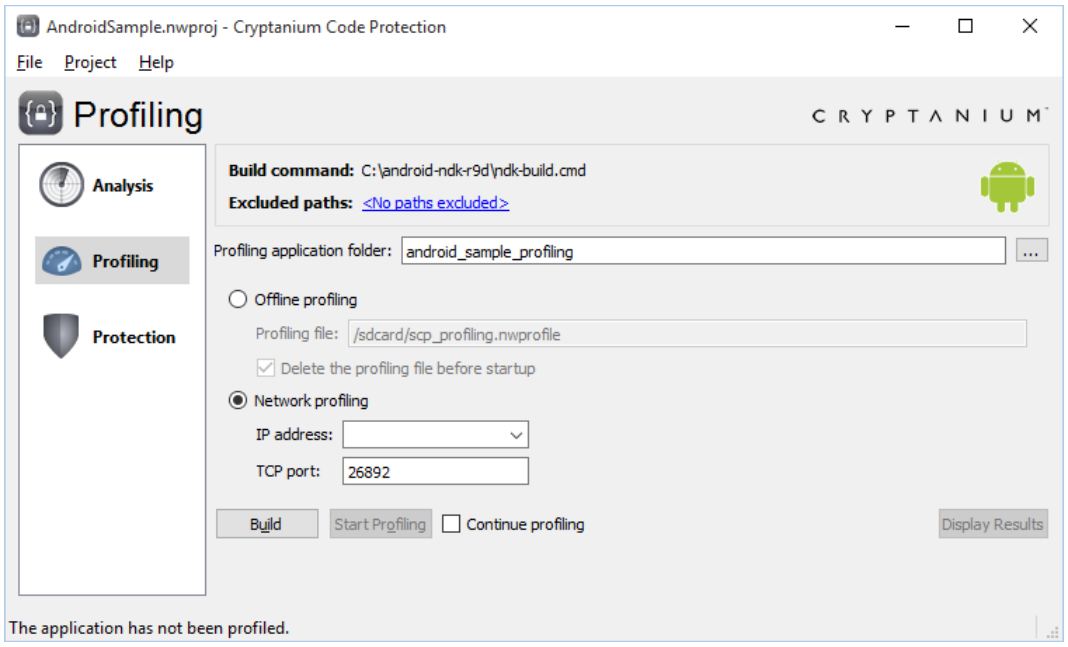
> ⚠ Profiling is optional, but we strongly recommend you to always profile your applications. If an application is not profiled, Code Protection will apply default protection to the entire code, which may result in slow execution speed. Profiling enables Code Protection to automatically adjust the security level and increase execution speed.

If your Code Protection project does not require any modifications, you can also execute the profiling step using the Command-Line Tool as described in §8.

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 the source code is modified.

4.1 Overview of the Profiling Page

The **Profiling** page provides controls for building the profiling edition of the application and executing the profiling process (see Figure 4.1).



Build command: C:\android-ndk-r9d\ndk-build.cmd
Excluded paths: <No paths excluded>
Profiling application folder: android_sample_profiling
( ) Offline profiling
Profiling file: /sdcard/scp_profiling.nwprofile
[✓] Delete the profiling file before startup
(●) Network profiling
IP address:
TCP port: 26892
[ ] Continue profiling
The application has not been profiled.

Figure 4.1: Profiling page

4.2 Executing the Profiling Step

To profile the application, proceed as follows:

1. Open the **Profiling** page.

2. Ensure the **Profiling application folder** field contains a valid folder name.

   This is the location where source code of the profiling application will be placed and built.

3. Execute one of the following profiling methods depending on your needs:

   Offline profiling (see §4.2.1)

   The profiling application collects and stores profiling statistics in a special file with extension `.nwprofile` on the device. It is then the developer's responsibility to load the file into Code Protection.

   Network profiling (see §4.2.2)

   The profiling application sends profiling statistics to Code Protection via network at run time.

   To use this method, the following conditions must be met:

   - The profiling application must be allowed to communicate over network.
   - The target device must be able to access the workstation where Code Protection is running over the network.
   - The TCP port used for communication between Code Protection and the profiling application should not be restricted.

4. Optionally, when the profiling process is successfully completed, click **Display Results** to see profiling statistics. For detailed information on the results displayed, see §4.3.

> ⚠ If you have built a static library, make sure the additional static libraries generated by Code Protection are linked into the final executable as well (see §6.1.1).

4.2.1 Offline Profiling

To profile the application using the offline profiling method, proceed as follows:

1. In the **Profiling** page, select the **Offline profiling** option.

2. In the **Profiling file** field, specify where the profiling statistics file must be saved on the target device and how it will be named.

   > ⚠ The specified location must be a writable folder.

   The location can be either a path relative to the profiling application executable, or an absolute path.

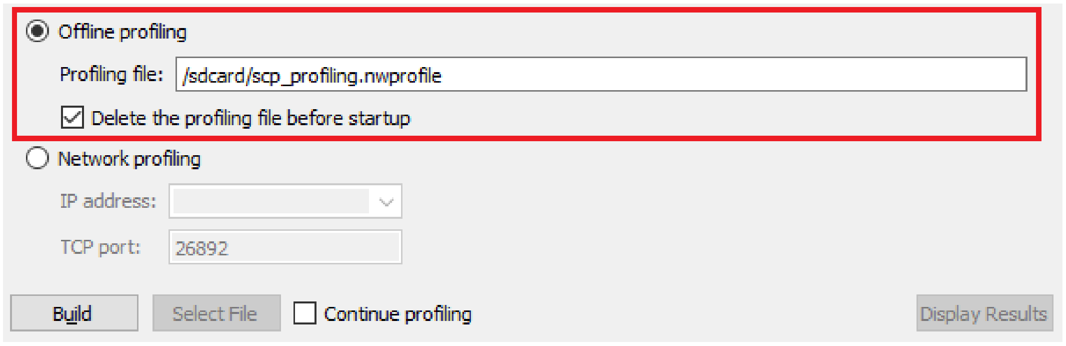
   The **Profiling file** field is not displayed for iOS applications because, on iOS devices, the only writable location of the profiling application is its **Documents** folder, which will be selected by default.

3. If you are building an Android application, and you selected the profiling statistics file to be saved in the external storage, add the following permissions to the `AndroidManifest.xml` file of the profiling application directly under main `<manifest>` tag (if not present):

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

4. Optionally, select the **Delete the profiling file before startup** check box (see Figure 4.2).

   If this check box is selected, the profiling application will create a new profiling statistics file every time it is started and overwrite any data gathered from previous runs. If the check box is not selected, new profiling statistics will be appended to existing data in the file.



● Offline profiling
Profiling file: /sdcard/scp_profiling.nwprofile
[✓] Delete the profiling file before startup
○ Network profiling
IP address:
TCP port: 26892
[ ] Continue profiling

Figure 4.2: Offline profiling

5. If you want to append the profiling statistics gathered from the profiling application to the data that is already stored in the profiling database, select the **Continue profiling** check box.

   If this check box is not selected, Code Protection will discard all previously gathered profiling data.

6. To build the profiling application, click **Build**.

   Code Protection invokes the build system and starts building the profiling application. A progress bar appears in the lower part of the page indicating the current build progress. For information on the progress bar, see §2.2.5.

7. When the profiling application build has completed, install the application on an appropriate target device.

8. If the device is running Android 6.0 (API 23) or later, and your application is targeting API 23 or later, and you have selected the profiling statistics file to be saved in the external storage, explicitly enable the permission for the profiling application to write to the external storage using one of the following methods:

   - Open global settings, select your profiling application, and, under **Permissions**, enable the permission to write to the external storage.
   - Grant the permission to read and write to the external storage from the command line using the `pm grant` command.
     For example, if you are using ADB, this can be achieved as follows:

     ```
     adb shell pm grant «package name» android.permission.READ_EXTERNAL_STORAGE
     adb shell pm grant «package name» android.permission.WRITE_EXTERNAL_STORAGE
     ```

     where *«package name»* is the name of your profiling application's package.

9. Run the profiling application on the device and work with it in a manner similar to real-life usage.

   Such behavior will provide realistic profiling statistics that Code Protection will use to try to achieve a good balance between speed and security. To obtain a more fine-tuned balance for your application, you can use code marking as described in §7.

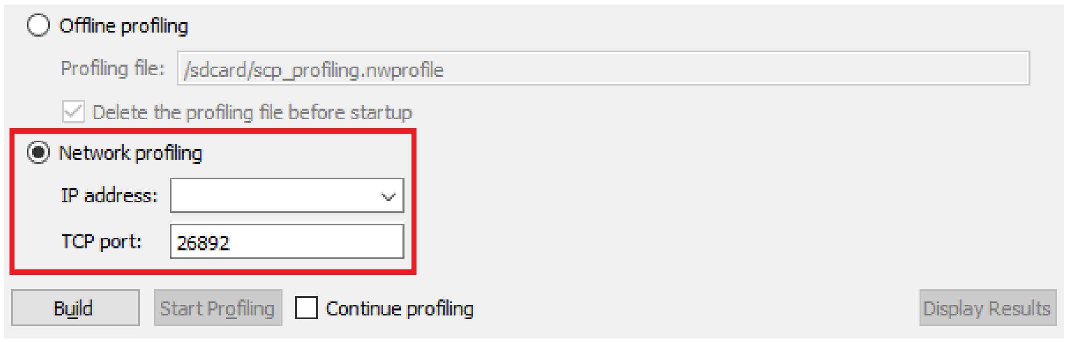
10. Stop the application after you have been working with it for sufficient time.
11. Locate the profiling statistics file on the device and transfer it to a place where Code Protection can access it.
12. Click **Select File** and select the profiling statistics file.

    As an indication of successfully completed profiling, Code Protection will display the most frequently called functions in the output field. You can see the output field by clicking **Show Details**.

4.2.2 Network Profiling

To profile the application using the network profiling method, proceed as follows:

1. In the **Profiling** page, select the **Network profiling** option (see Figure 4.3).



○ Offline profiling
Profiling file: /sdcard/scp_profiling.nwprofile
✓ Delete the profiling file before startup
● Network profiling
IP address:
TCP port: 26892

Figure 4.3: Network profiling

2. Provide information in the following fields:

   IP address

   IP address that will be embedded in the profiling application. The application will try to connect to this address once it is started on the target device.
   You can override this IP address by creating an environment variable named `NW_PROFILE_IP` on the device where the profiling application will be run. The value of this variable must be the required IP address.

   TCP port

   TCP port that the profiling application will use to connect to Code Protection.

3. If you want to append the profiling statistics gathered from the profiling application to the data that is already stored in the profiling database, select the **Continue profiling** check box.

   If this check box is not selected, Code Protection will discard all previously gathered profiling data.

4. To build the profiling application, click **Build**.

   Code Protection invokes the build system and starts building the profiling application. A progress bar appears in the lower part of the page indicating the current build progress. For information on the progress bar, see §2.2.5.

5. When the profiling application build has completed, install the application on an appropriate target device.

6. To start the Code Protection profiler, click **Start Profiling**.

   Now Code Protection is ready to accept connections from the profiling application. If you click **Show Details**, the output field displays the following information:

   ```
   Profiling - resetting function profiling information...
   Profiling - listening on port 26892...
   ```

7. Run the profiling application on the target device.

   A successful connection to Code Protection is indicated by the following message in the output field:

   ```
   [CONNECTION #1 FROM 192.168.0.102:54164]
   ```

   The example message above means that the profiling application running on a device with IP address 192.168.0.102 has successfully established the first connection to Code Protection and is transferring profiling information. The profiling application may establish additional connections if it starts additional processes. Each new process will create a new connection to Code Protection. Code Protection combines profiling information together from all simultaneous connections.

8. Work with the profiling application in a manner similar to real-life usage.

   Such behavior will provide realistic profiling statistics that Code Protection will use to try to achieve good balance between speed and security. To obtain a more fine-tuned balance for your application, you can use code marking as described in §7.

9. After working with the profiling application for sufficient time, stop the application on the device and make sure all its processes are terminated.

   An ended connection will be identified by the following message in the **Output** field:

   ```
   [#1 DISCONNECTED]
   ```

   The Code Protection profiler stops automatically once all established connections are disconnected.

   As an indication of successfully completed profiling, Code Protection will display the most frequently called functions in the output field. You can see the output field by clicking **Show Details**.

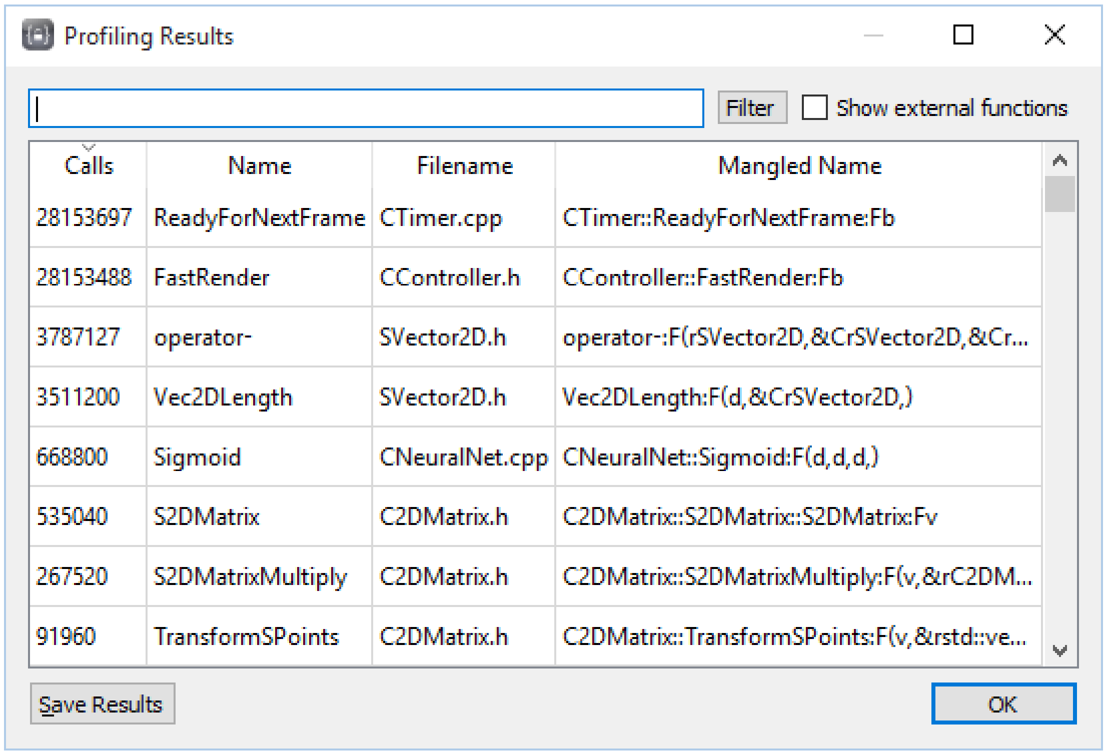
4.3 Viewing Profiling Results

Once you have successfully executed the profiling process, you can view specific profiling statistics about individual functions in your application. This information is available in the **Profiling Results** window.

To view profiling results, proceed as follows:

1. In the **Profiling** page, click **Display Results**.

   The **Profiling Results** window is displayed (see Figure 4.4).



Calls	Name	Filename	Mangled Name
28153697	ReadyForNextFrame	CTimer.cpp	CTimer::ReadyForNextFrame:Fb
28153488	FastRender	CController.h	CController::FastRender:Fb
3787127	operator-	SVector2D.h	operator-:F(rSVector2D,&CrSVector2D,&Cr...
3511200	Vec2DLength	SVector2D.h	Vec2DLength:F(d,&CrSVector2D,)
668800	Sigmoid	CNeuralNet.cpp	CNeuralNet::Sigmoid:F(d,d,d,)
535040	S2DMatrix	C2DMatrix.h	C2DMatrix::S2DMatrix::S2DMatrix:Fv
267520	S2DMatrixMultiply	C2DMatrix.h	C2DMatrix::S2DMatrixMultiply:F(v,&rC2DM...
91960	TransformSPoints	C2DMatrix.h	C2DMatrix::TransformSPoints:F(v,&rstd::ve...

Figure 4.4: Profiling results

The table displayed lists all functions detected in the last profiling run. The data is organized into the following columns:

- **Calls**: how many times the function was called during the profiling phase
- **Name**: function name
- **Filename**: source file where this function is defined
- **Mangled Name**: mangled function name

Initially, only functions within the target application's source code are displayed, but you can also display functions included from external sources (for example, those in the `std` namespace) by selecting the **Show external functions** check box.

2. Optionally, to control the way the table is displayed, do the following actions as necessary:
   - To find a specific function or limit the number of displayed functions, enter a name in the filter field and click **Filter**.
     The entered name will be matched against text in the **Name** and **Filename** columns.
   - To sort the table by a specific column, click the corresponding column heading.
     The sorting order is represented by a small arrow above the column name.
3. Optionally, to save profiling results as a CSV file, click **Save Results**.

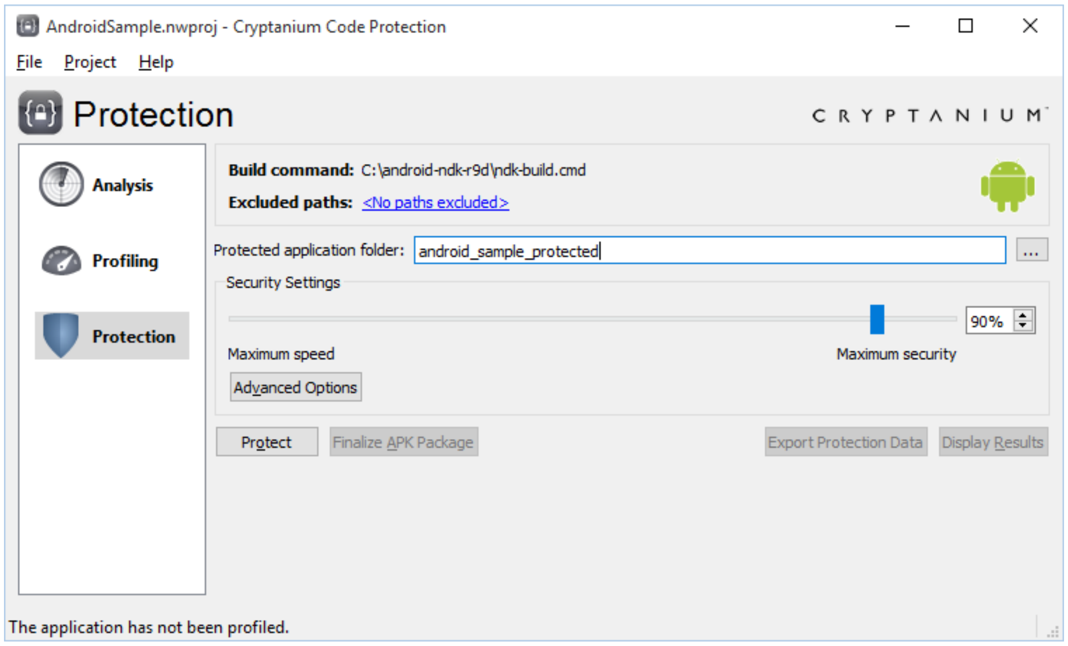
5 Building the Protected Application

This chapter describes how to use the GUI to build a protected version of your application, which is the last step in the overall protection process (see §1.2.2).

If your Code Protection project does not require any modifications, you can also build the protected application using the Command-Line Tool as described in §8.

5.1 Overview of the Protection Page

The **Protection** page (see Figure 5.1) is where you configure security settings and build the final protected edition of the application.



Build command: C:\android-ndk-r9d\ndk-build.cmd
Excluded paths: <No paths excluded>
Protected application folder: android_sample_protected
Security Settings
Maximum speed 90% Maximum security
The application has not been profiled.

Figure 5.1: Protection page

5.2 Executing the Protection Step

To build the protected application, proceed as follows:

1. If necessary, in the **Protected application folder** field, select the folder where the final protected application will be placed.
2. Optionally, adjust the security slider.

   This slider allows you to set the general security level of the application. Higher number means stronger security but slower execution speed and larger binary size. The slider has a direct impact on other Code Protection security parameters. Namely, adjusting the slider will proportionally increase or decrease the weight of sliders in the **Advanced Options** window (see §5.3).
3. Optionally, to configure advanced security options, click **Advanced Options** and follow the

instructions in §5.3.

4. To build the final protected application, click **Protect**.

   Code Protection invokes the build system and starts building the protected application. A progress bar appears in the lower part of the page indicating the current build progress. For information on the progress bar, see §2.2.5.

   For information on the **Finalize APK Package** button, see §12.

5. If you are protecting a static library, click **Export Protection Data** to obtain the protection data file that will be required to build the final executable.

   Also, make sure the additional static libraries generated by Code Protection (if there are any) are linked into the final executable as well (see §6.1.1).

   For more information on working with protected static libraries, see §6.

6. Optionally, to see specific details and statistics about the protected application, click **Display Results**.

   For detailed information on the protection results displayed, see §5.4.

5.3 Configuring Advanced Security Options

Code Protection gives you full control over the security features applied to the protected application. You can enable and disable most of the features and adjust their security level. All these operations can be performed in the **Advanced Options** window.

To open the **Advanced Options** window where you can configure advanced security options, open the **Protection** page and click **Advanced Options** as shown in Figure 5.2.

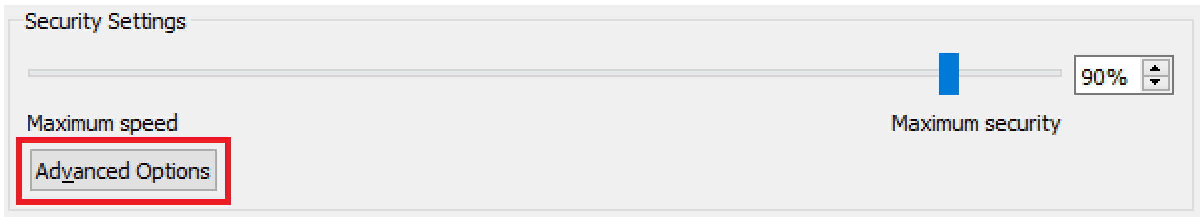
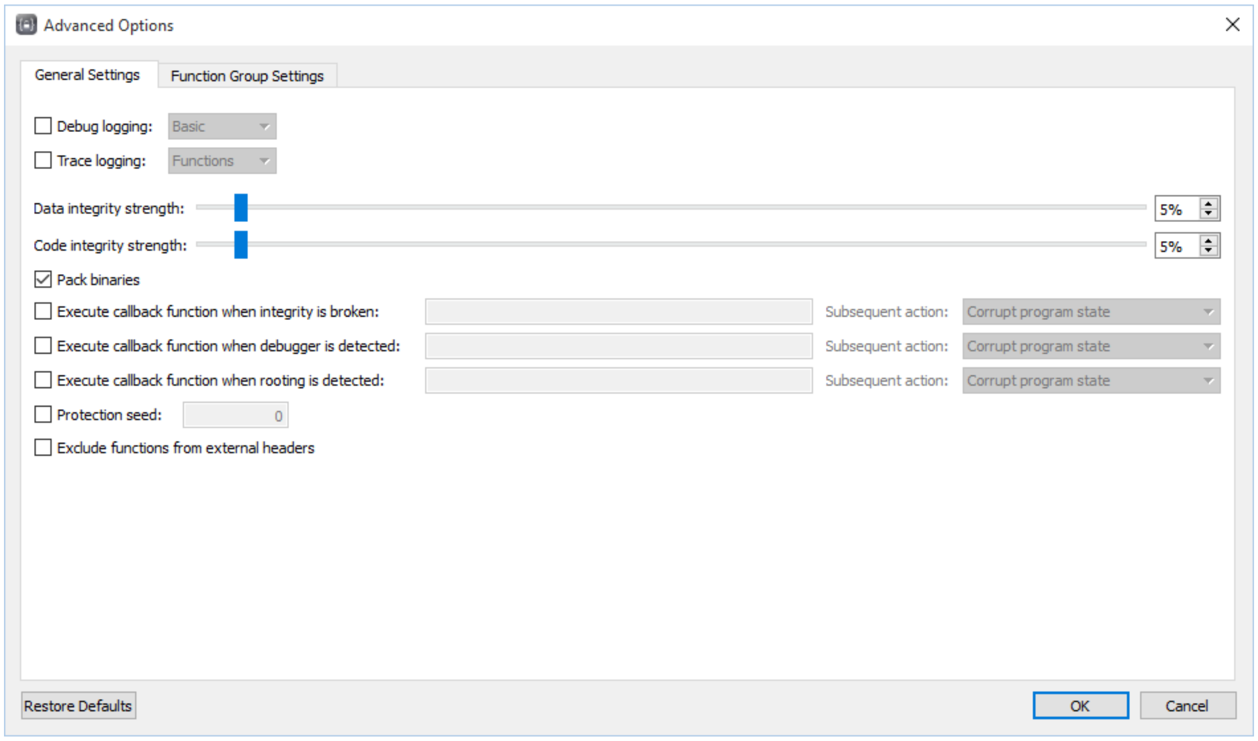


Figure 5.2: "Advanced Options" button

Controls in the **Advanced Options** window are divided into two tabs that are described in the following subsections.

5.3.1 "General Settings" Tab

The **General Settings** tab (see Figure 5.3) contains security settings that are equally applied to the entire application.



- [ ] Debug logging: Basic
- [ ] Trace logging: Functions

Data integrity strength: 5%

Code integrity strength: 5%

- [✓] Pack binaries
- [ ] Execute callback function when integrity is broken: — Subsequent action: Corrupt program state
- [ ] Execute callback function when debugger is detected: — Subsequent action: Corrupt program state
- [ ] Execute callback function when rooting is detected: — Subsequent action: Corrupt program state
- [ ] Protection seed: 0
- [ ] Exclude functions from external headers

Figure 5.3: General settings

The following are the settings in this tab.

“Debug logging” check box

If selected, the protected application will write Code Protection related debug information to the console. The following additional options are available when this check box is selected:

- **Basic**: The application will write debug information only about attempted attacks. If an integrity, anti-debug, jailbreak, rooting, or method swizzling check is triggered, the application will write the information just before terminating the execution.
- **Verbose**: The application will write all debug information about Code Protection activities, such as verifying checksums or starting or ending anti-debug, jailbreak, rooting, and method swizzling checks.

⚠ The **Debug logging** option is intended only for development and testing. It must never be enabled in the final production application as it can provide valuable information to potential hackers.

⚠ If the protected application is configured to corrupt the program state in case a threat is detected (such as tampering, debugging, jailbreak, rooting, or method swizzling), program state corruption will not be executed if debug logging is enabled.

“Trace logging” check box

If selected, the protected application will write execution flow related information to the console. The following additional options are available when this check box is selected:

- **Functions**: The application will write a message to the console every time execution enters or leaves a function.
- **Basic blocks**: This option will produce the same output as the **Functions** option, but it will additionally write a message whenever execution enters a code block.

⚠ The **Trace logging** option is intended only for development and testing. It must never be enabled in the final production application as it can provide valuable information to potential hackers.

“Data integrity strength” slider

This slider allows you to set the balance between stronger data integrity protection (see §1.2.3.1) and faster execution speed. A higher value means that each data integrity checksum will cover a larger portion of data, resulting in more overlaps with data portions covered by other checksums, but the execution speed of the application will be slower.

“Code integrity strength” slider

This slider allows you to set the balance between stronger code integrity protection (see §1.2.3.1) and faster execution speed. A higher value means that each code integrity checksum will cover a larger portion of code, resulting in more overlaps with code portions covered by other checksums, but the execution speed of the application will be slower.

“Pack binaries” check box (Windows, Android)

If selected, Code Protection will encrypt all executable binaries of the application to prevent static analysis. For Android applications, only libraries written in the native C/C++ code will be encrypted.

For more information on this feature, see §1.2.3.12.

“Strip debug symbols” check box (OS X, iOS, Linux)

If selected, Code Protection strips debug symbols from the executable. Debug symbols may provide valuable information to potential attackers.

Visual Studio and Android NDK strip debug symbols by default. That is why this option is not displayed for Windows and Android applications.

⚠ We recommend to always keep this check box selected.

“Execute callback function when integrity is broken” check box

If selected, the protected application will execute a custom callback function when code or data tampering is detected. For more information on this feature, see §1.2.3.17.

You must provide the name of the callback function in the text field next to the check box. This function must be in the global namespace and have C linkage. The following is an example of a very simple callback function:

```
#ifdef __cplusplus
extern "C" {
#endif
void TamperCallback() {
  printf("Tampering detected!\n");
}
#ifdef __cplusplus
}
#endif
```

Selecting this check box also enables the **Subsequent action** list box, which is described further down.

“Execute callback function when debugger is detected” check box

Same as the **Execute callback function when integrity is broken** check box, but the specified function is called when a debugger is detected.

“Execute callback function when jailbreak is detected” check box (iOS)

Same as the **Execute callback function when integrity is broken** check box, but the specified function is called when a jailbroken iOS device is detected.

“Execute callback function when swizzling is detected” check box (iOS)

Same as the **Execute callback function when integrity is broken** check box, but the specified function is called when method swizzling is detected.

“Execute callback function when rooting is detected” check box (Android)

Same as the **Execute callback function when integrity is broken** check box, but the specified function is called when a rooted Android device is detected.

“Subsequent action” list box

Allows you to specify what should happen after the corresponding callback function is executed.

The following options are available:

- **Corrupt program state**: Corrupts the program state, which eventually leads to application crash. This is the default action when calling of a callback function is not enabled.
- **Continue execution**: Leaves the application running. You should be very careful when using this option because it gives more freedom to the attacker.

“Use protection seed” check box

If selected, you can specify an arbitrary integer in the provided field. This integer will be used as the seed for generating the protected code. The same seed will always produce the same binary footprint of the protected application.

If this check box is not selected (recommended setting), Code Protection will generate code with random binary footprint each time the protection process is run. This mechanism adds the diversification feature to your application (see §1.2.3.18).

“Exclude functions from external headers” check box

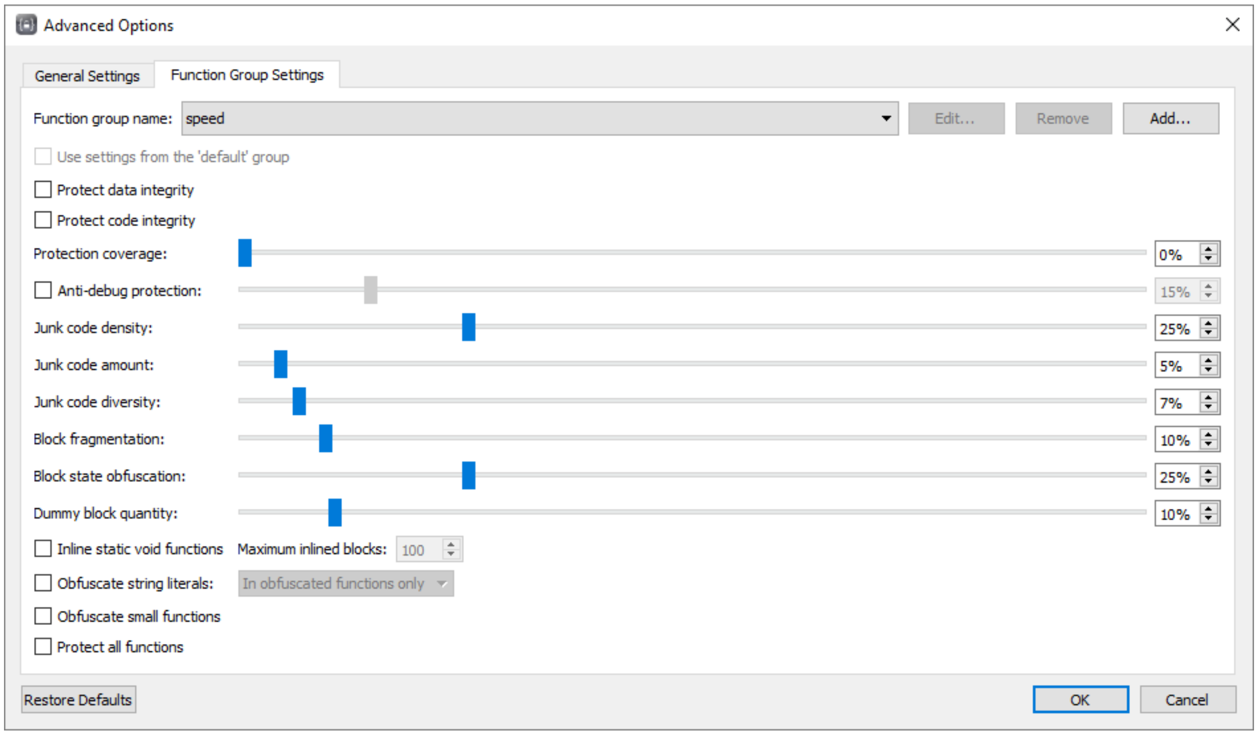
If selected, Code Protection will not process functions from headers that are located outside the source root folder.

For example, if you enable this option, Code Protection will not protect functions in the standard C/C++ libraries.

If required, you can click **Restore Defaults** to reset all settings in the **General Settings** window to default values. This operation does not affect settings in the **Function Group Settings** tab.

5.3.2 “Function Group Settings” Tab

The **Function Group Settings** tab (see Figure 5.4) contains security settings that can be separately adjusted for each individual function group. For information on defining function groups in the source code, see §7.3.



Function group name: speed

☐ Use settings from the 'default' group
☐ Protect data integrity
☐ Protect code integrity

Protection coverage: 0%
☐ Anti-debug protection: 15%
Junk code density: 25%
Junk code amount: 5%
Junk code diversity: 7%
Block fragmentation: 10%
Block state obfuscation: 25%
Dummy block quantity: 10%

☐ Inline static void functions Maximum inlined blocks: 100
☐ Obfuscate string literals: In obfuscated functions only
☐ Obfuscate small functions
☐ Protect all functions

Figure 5.4: Function group settings

All setting values displayed in this tab apply only to the function group currently selected in the **Function group name** list box. Different function groups may have different setting values.

Initially, only the predefined function groups (`default`, `speed`, `security`, and `obfuscation`) are available for selection in the **Function group name** list box. If you want to adjust security settings for a custom function group that you defined in the source code, you must first manually add the group name in the GUI.

The following operations are available for managing function groups in the **Function Group Settings** tab:

- To add configuration for a new function group, click **Add** and enter its name (exactly as it is used in the source code) in the pop-up window. Then you can select this group in the **Function group name** list box and adjust the security settings as needed.
- To remove a function group from the project settings, select it in the list box and click **Remove**. The predefined function groups cannot be removed. This operation will not remove the function group code markers from source code.
- To rename a function group, select it in the **Function group name** list box, click **Edit**, and modify its name in the pop-up window. This operation will not rename the function group code markers in the source code. You should do this only if you have manually changed the markers in the source code.

The following are the security settings in the **Function Group Settings** tab:

“Use settings from the ’default’ group” check box

If selected, security settings applied to the selected function group will be the same as for the **default** function group.

“Protect data integrity” check box

If selected, data integrity checks (see §1.2.3.1) will be embedded in functions of the selected function group.

> ⚠ Disabling this option will significantly reduce the security of your application. However, we do not recommend enabling this setting if the **Protect code integrity** check box is disabled.

“Protect code integrity” check box

If selected, code integrity checks (see §1.2.3.1) will be embedded in functions of the selected function group.

> ⚠ Disabling this option will significantly reduce the security of your application.

“Protection coverage” slider

This parameter controls how many functions are protected with Code Protection security features, such as obfuscation, integrity checks, anti-debug checks, jailbreak checks, rooting checks, and method swizzling checks.

By raising the value you will increase security, but the execution speed will be slower, and the binary size will be larger.

“Anti-debug protection” check box and slider

If the check box is selected, anti-debug checks will be embedded in the protected code (see §1.2.3.3).

The slider allows you to set the balance between stronger anti-debug protection and faster execution speed.

> ⚠ Disabling this option will significantly reduce the security of your application.

“Anti-jailbreak protection” check box and slider (iOS)

If the check box is selected, jailbreak detection will be embedded in the protected code (see §1.2.3.4).

The slider allows you to set the balance between stronger jailbreak protection and faster execution speed.

> ⚠ Disabling this option will significantly reduce the security of your application.

"Anti-swizzling protection" check box and slider (iOS)

If the check box is selected, method swizzling detection will be embedded in the protected code (see §1.2.3.5).

The slider allows you to set the balance between stronger method swizzling protection and faster execution speed.

⚠ Disabling this option will significantly reduce the security of your application.

"Anti-rooting protection" check box and slider (Android)

If the check box is selected, rooting detection will be embedded in the protected code (see §1.2.3.6).

The slider allows you to set the balance between stronger rooting protection and faster execution speed.

⚠ Disabling this option will significantly reduce the security of your application.

"Junk code density" slider

This parameter specifies how frequently junk code statements are inserted into the source to confuse the attacker. This junk code is executable but it does not really do anything.

By raising the value you will increase security, but the code size will be larger and execution speed will be slower.

"Junk code amount" slider

This parameter specifies the size of junk code statements inserted into the source.

By raising the value you will increase security, but the code size will be larger and execution speed will be slower.

"Junk code diversity" slider

This parameter controls the quantity of variables used in the junk code. The more variables are used, the more diverse the junk code statements appear to the attacker.

By raising the value you will increase security, but the amount of stack used at run time will also be larger.

"Block fragmentation" slider

In the source, linear code, which does not contain any conditional statements, is put into one code block. Code Protection splits code blocks into smaller chunks to generate complex flattened code. This slider specifies the size of code block chunks. A higher number will generate smaller chunks.

By raising the value you will increase security, but the code size will be larger and execution speed will be slower.

"Block state obfuscation" slider

This parameter specifies how frequently Code Protection obfuscates the code used to calculate the transition from one code block to another.

By raising the value you will increase security, but the code size will be larger and execution speed will be slower.

“Dummy block quantity” slider

For increased obfuscation, Code Protection makes copies of certain blocks of valid code and adds them to the source. Such dummy blocks will appear as ordinary code to the attacker, but they will never be executed. This slider affects the number of dummy blocks inserted in code.

By raising the value you will increase security, but the code size will be larger.

“Inline static void functions” check box

If selected, Code Protection will try to inline static void functions into the calling function (see §1.2.3.13).

The **Maximum inlined blocks** setting below the check box allows you to set the amount of code that will be inlined in the calling function.

By raising the value you will increase security, but the binary size will be larger.

“Obfuscate Objective-C message calls” list box (OS X, iOS)

Provides the following options that control how Objective-C message call obfuscation (see §1.2.3.14) is applied to the application’s source code:

- **None**: disables Objective-C message call obfuscation

  ⚠ This value will significantly reduce the security of your application.

- **In obfuscated functions only** (default value): obfuscates message calls only in obfuscated functions
- **In all functions**: obfuscates message calls in all functions, not just the obfuscated ones

Objective-C message call obfuscation improves OS X and iOS code security but reduces execution speed.

“Obfuscate string literals” list box

Provides the following options that control how string literal obfuscation (see §1.2.3.16) is applied to the application’s source code:

- **None**: disables string literal obfuscation
- **In obfuscated functions only** (default value): obfuscates string literals only in obfuscated functions
- **In all functions**: obfuscates string literals in all functions, not just the obfuscated ones

String literal obfuscation improves security but reduces execution speed and increases the binary size.

“Obfuscate small functions” check box

If selected, Code Protection will apply code obfuscation even to small functions. This option may be useful for function groups that contain especially sensitive functions.

By selecting this option you will increase security, but the binary size will be larger.

“Protect all functions” check box

If selected, Code Protection will protect all functions in the group regardless of profiling results (except those functions that are explicitly excluded from protection). Also, if this check box is selected, the slider settings of the selected group will completely ignore the global security slider in the **Protection** page.

If required, you can click **Restore Defaults** to reset the settings of the currently selected function group to default values.

5.4 Viewing Protection Results

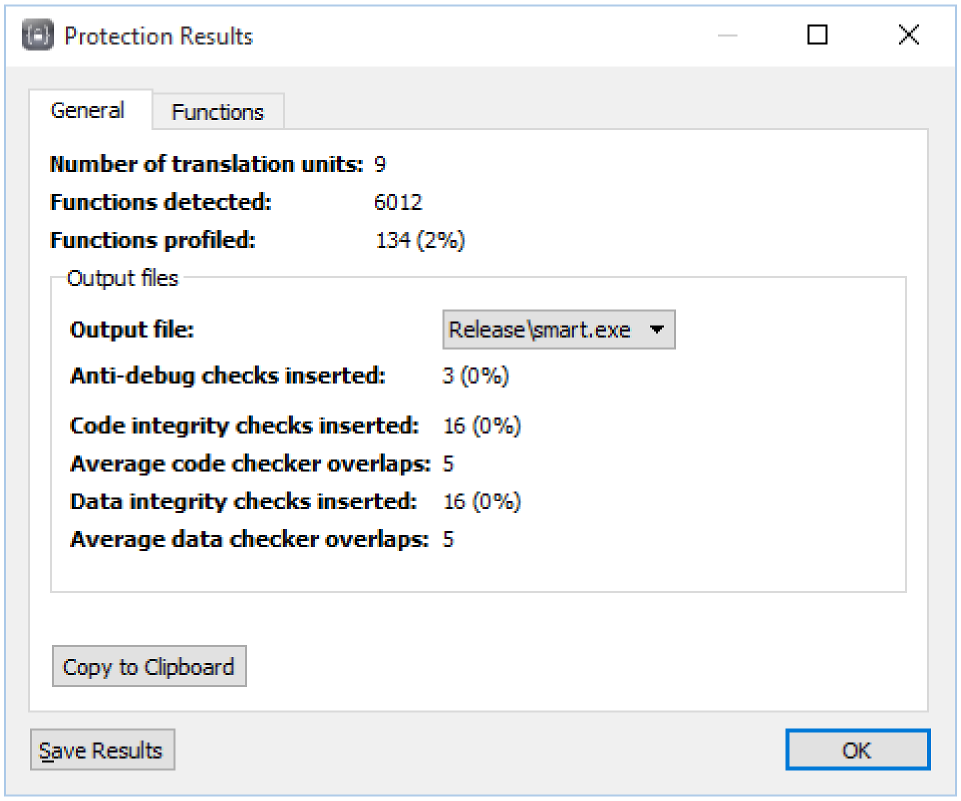
Once the protected version of an application is built, you can view specific details and statistics about the applied protection. This information is available in the **Protection Results** window.

To view protection results, in the **Protection** page, click **Display Results**.

The **Protection Results** window is displayed. Information in this window is divided into two tabs described in the following subsections.

5.4.1 "General" Tab

The **General** tab (see Figure 5.5) displays overall information about the protected application.



Number of translation units:	9
Functions detected:	6012
Functions profiled:	134 (2%)
Output file:	Release\smart.exe
Anti-debug checks inserted:	3 (0%)
Code integrity checks inserted:	16 (0%)
Average code checker overlaps:	5
Data integrity checks inserted:	16 (0%)
Average data checker overlaps:	5

Figure 5.5: General statistics

The following parameters are displayed:

Number of translation units

number of protected translation units in the application

Functions detected

number of functions detected and processed

Functions profiled

number of functions called during the profiling phase

Output file

allows you to select individual compiled output files and see their protection statistics

Anti-debug checks inserted

number of anti-debug checks inserted into the file selected in the **Output file** list box

Rooting checks inserted (Android)

number of rooting detection checks compiled into the file selected in the **Output file** list box

Code integrity checks inserted

number of code integrity checkers inserted into the file selected in the **Output file** list box

Average code checker overlaps

average number of code integrity checks protecting any arbitrary byte in the part of memory where the executable code section of the file selected in the **Output file** list box is loaded

Data integrity checks inserted

number of data integrity checkers inserted into the file selected in the **Output file** list box

Average data checker overlaps

average number of data integrity checks protecting any arbitrary byte in the part of memory where the read-only data of the file selected in the **Output file** list box is loaded

Jailbreak checks inserted (iOS)

number of jailbreak detection checks compiled into the file selected in the **Output file** list box

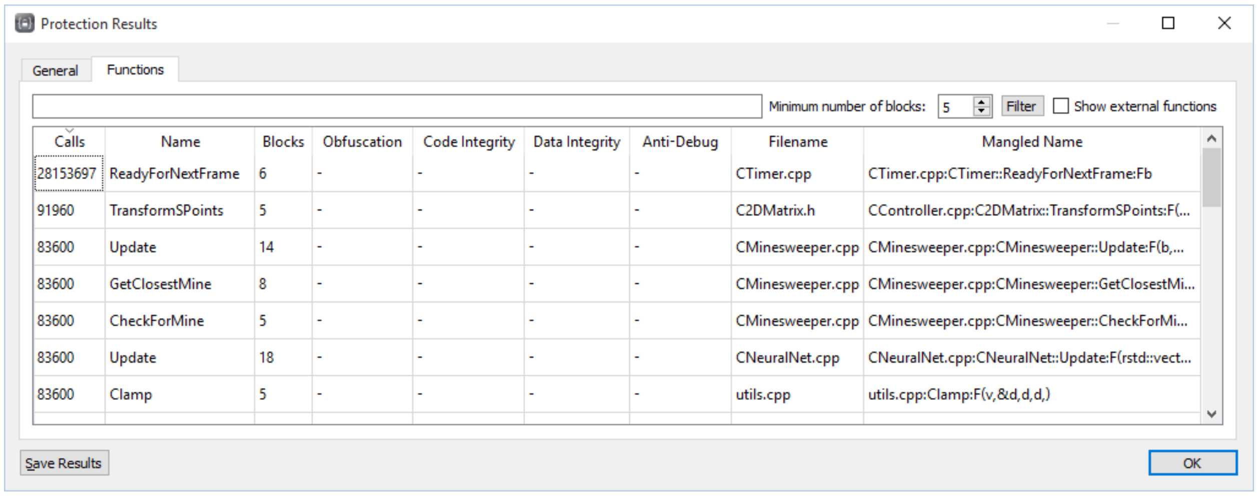
Method swizzling checks inserted (iOS)

number of method swizzling detection checks compiled into the file selected in the **Output file** list box

The percentage value displayed in the brackets is calculated based on the **Functions detected** number.

5.4.2 "Functions" Tab

The **Functions** tab (see 5.6) lists all processed functions in the protected application, and shows what security features were applied to each one of them.



Calls	Name	Blocks	Obfuscation	Code Integrity	Data Integrity	Anti-Debug	Filename	Mangled Name
28153697	ReadyForNextFrame	6	-	-	-	-	CTimer.cpp	CTimer.cpp:CTimer::ReadyForNextFrame:Fb
91960	TransformSPoints	5	-	-	-	-	C2DMatrix.h	CController.cpp:C2DMatrix::TransformSPoints:F(...
83600	Update	14	-	-	-	-	CMinesweeper.cpp	CMinesweeper.cpp:CMinesweeper::Update:F(b,...
83600	GetClosestMine	8	-	-	-	-	CMinesweeper.cpp	CMinesweeper.cpp:CMinesweeper::GetClosestMi...
83600	CheckForMine	5	-	-	-	-	CMinesweeper.cpp	CMinesweeper.cpp:CMinesweeper::CheckForMi...
83600	Update	18	-	-	-	-	CNeuralNet.cpp	CNeuralNet.cpp:CNeuralNet::Update:F(rstd::vect...
83600	Clamp	5	-	-	-	-	utils.cpp	utils.cpp:Clamp:F(v,&d,d,d,)

Figure 5.6: Function statistics

The information is organized into the following columns:

Calls
: how many times the function was called in the last profiling run

Name
: function name

Obfuscation
: "YES" indicates that code obfuscation was applied to this function (see §1.2.3.2)

Code Integrity
: "YES" indicates that code integrity protection was applied to this function (see §1.2.3.1)

Data Integrity
: "YES" indicates that data integrity protection was applied to this function (see §1.2.3.1)

Anti-Debug
: "YES" indicates that anti-debug checks were inserted in this function (see §1.2.3.3)

Jailbreak Detection (iOS)
: "YES" indicates that jailbreak detection checks were inserted in this function (see §1.2.3.4)

Swizzling Detection (iOS)
: "YES" indicates that method swizzling detection checks were inserted in this function (see §1.2.3.5)

Rooting Detection (Android)
: "YES" indicates that rooting detection checks were inserted in this function (see §1.2.3.6)

Filename
: source file where this function is defined

Mangled Name

mangled function name

Initially, only functions within the target application's source code are displayed, but you can also display functions included from external sources (for example, those in the `std` namespace) by selecting the **Show external functions** check box.

Optionally, to control the way the table is displayed, do the following actions as necessary:

- To find a specific function or limit the number of displayed functions, enter a name in the filter field and click **Filter**.

  The entered name will be matched against text in the **Name** and **Filename** columns.

- To sort the table by a specific column, click the corresponding column heading.

  The sorting order is represented by a small arrow above the column name.

To export protection data to a CSV file, click **Save Results**.

6 Protecting Static Libraries

Code Protection can be used to increase the protection of both executable applications, and static libraries, which will later be compiled into other (potentially, unprotected) executables. This chapter focuses on how to correctly create protected static libraries with Code Protection and how to compile them into the final executable.

6.1 Building Static Libraries Using Code Protection

In general, the process of building a static library using Code Protection is the same as building a stand-alone executable application — you create a Code Protection project, analyze the source code of the library, optionally profile it using an executable that is linked with the library, and finally build the protected version of the library (see §1.2.2).

However, the following additional steps must be completed when building static libraries with Code Protection:

- When Code Protection builds the profiling version or protected version of a static library, it may also generate one or several additional static libraries, depending on project settings and the target architecture. If these additional libraries are generated, they must also be linked into the final executable. For details on this point, see §6.1.1.
- The protected version of a static library cannot be simply compiled into an executable because the libraries contain integrity checksums that must be updated after the final application is built (unless code and data integrity is switched off as described in §5.3.2). Without this updating, the final application will behave as if integrity checks are failing. For details on this point, see §6.1.2.
- If you are protecting a static library for OS X or iOS and you have the Mach-O binary signature verification feature enabled (see §1.2.3.7), you may not know which applications will use your library and what the developers' public keys of those applications will be. Therefore, in the Code Protection project settings, you may need to specify a sample application whose signature matches the necessary public key as described in §13.2.

6.1.1 Generated Static Libraries

When Code Protection builds the profiling version or the protected version of a static library, one or several additional static libraries may get generated, depending on project settings and the target architecture (no such libraries are generated in the analysis step). These libraries contain Code Protection specific internal functions necessary for the final executable to work. These generated libraries must also be linked into the final executable. If this is not done, the final executable will fail to build.

To find out if there are any additional libraries generated by Code Protection, you must look at the build output after the profiling or protected version of the static library is built. The identifying text to look for is the following:

```
WARNING! When you build the final executable, you will have to link «file name of the
library or libraries» as well.
```

This text will be followed by a path to the generated libraries.

The presence of these messages is an indication that you have to link the additional static libraries into the final executable as well.

6.1.2 Protection Data File

Code Protection requires a special protection data file to be provided for every protected static library that is included in the final application. These protection data files are necessary to update integrity checksums in the final application after it is built.

There are two ways how the protection data files can be obtained:

- Click the **Export Protection Data** button in the **Protection** page when the protected static library is built (see §5.2) as shown in Figure 6.1.

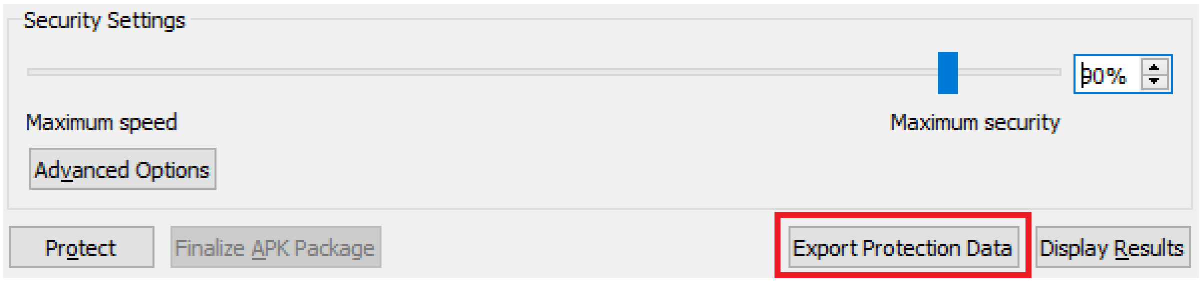


Figure 6.1: Exporting the protection data file

- Execute the Command-Line Tool with the "`--export-protection`" command (see §8).

As a result, you will get the protection data file, which has the extension `.nwdb`.

⚠ You must make sure the protection data file is always placed next to the protected static library and has the same name as the library.

6.2 Building the Final Executable with Code Protection

If the final application, which uses the protected static library, is also protected by Code Protection, you have to make sure the following conditions are met:

- Any additional static libraries, which were generated by Code Protection when the protected static library was built, must also be linked into the final executable.
- The protection data file (`*.nwdb`) must be placed next to the built static library and have the same name as the library. For example, if you have a protected static library named `MyLib.a` then the corresponding protection data file must be named `MyLib.nwdb` and you must place it next to the `MyLib.a` file. Then Code Protection will automatically detect and process the protection data file.

6.3 Building the Final Executable without Code Protection

If the final application, which uses the protected static library, is not itself protected by Code Protection, you have to make sure the following conditions are met:

- Any additional static libraries, which were generated by Code Protection when the protected static library was built, must also be linked into the final executable.
- You have to use a special utility named Binary Update Tool to process the information from the protection data file. This utility must be executed on the final executable when it is built in order to update the integrity checksums of the embedded static library. Subsequent subsections provide more details on the Binary Update Tool and how it should be used.

6.3.1 Binary Update Tool

Code Protection provides a stand-alone utility called the Binary Update Tool, which is used when you have a Code Protection protected static library (or several libraries) that is compiled into an application that itself is not protected by Code Protection. In that case, the Binary Update Tool must be executed on the final compiled application to update the embedded integrity checksums.

As an input, the Binary Update Tool requires the following:

- compiled binary application (containing the protected static library)
- protection data file (with extension `*.nwdb`) generated by Code Protection (see §6.1.2)

Figure 6.2 provides an overview of the input files of the Binary Update Tool and how they are obtained.

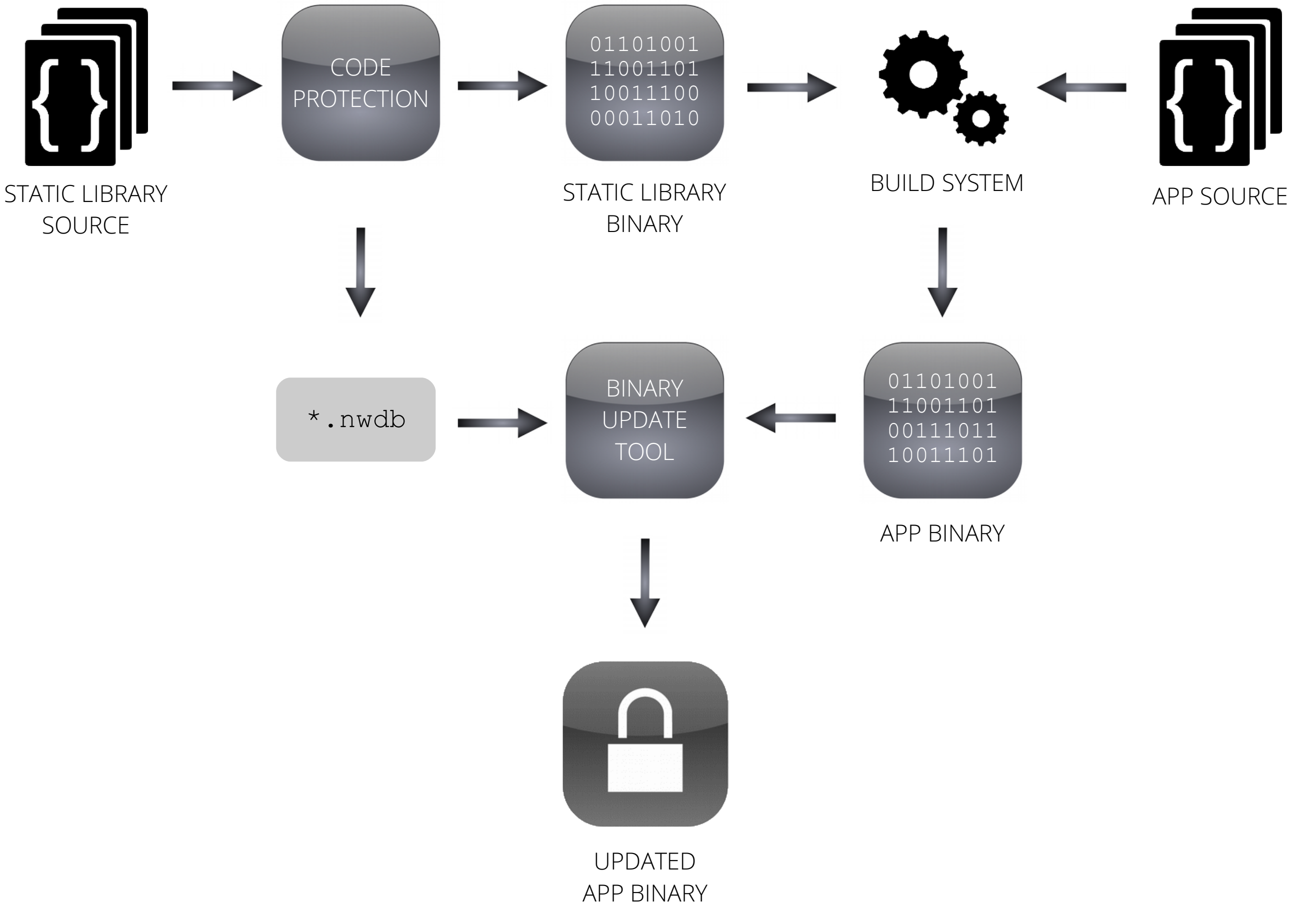


Figure 6.2: Compiling a protected static library into an unprotected application

If you build a protected static library that you intend to deliver to third parties (your customers) who do not have Code Protection, you must also give them the Binary Update Tool and all the necessary protection data files along with the protected static library.

⚠ OS X and iOS applications must be re-signed after running the Binary Update Tool, because the binary footprint will be different. You can perform signing using the **`codesign`** tool.

6.3.2 Running the Binary Update Tool

For Windows and Linux development platforms, the Binary Update Tool is provided as the archive `scp-update-binary.zip`, which contains the utility executable along with the libraries it requires. For the OS X development platform, the Binary Update Tool is delivered as a stand-alone executable `scp-update-binary`. These files are located in the Code Protection installation folder.

To process an executable with the Binary Update Tool, execute the following command:

```
scp-update-binary --binary="«compiled executable»" «*.nwdb file(s)»
```

"`--binary`" specifies a path to the application executable that contains the static library protected by Code Protection.

As the final parameter, you must provide one or several protection data files (separated by spaces), which correspond to the included protected static libraries. They can have any arbitrary names.

To find out the version number of the Binary Update Tool, execute the following command:

```
scp-update-binary --version
```

6.4 Delivering the Protected Static Library to Third Parties

If you have built a protected static library that will be used by third parties for building their own applications, you have to provide the following deliveries to them:

Additional libraries generated by Code Protection (if there are any)

If Code Protection generated any additional libraries when building the protected static library (see §6.1.1), provide these libraries as well and instruct third parties to link them into their final executables.

Binary Update Tool

The `scp-update-binary.zip` file must be provided if the final application will be built on Windows or Linux, or the `scp-update-binary` executable if the final applications will be built on OS X. Besides the additional libraries included in the `scp-update-binary.zip` file, system requirements for the Binary Update Tool are the same as for Code Protection (see §1.3.1). You must instruct third parties to execute the Binary Update Tool on the binary of the final executable.

7 Using Code Markers

This chapter describes how you can manually mark the source code to facilitate profiling and improve execution speed and security of the final protected application.

7.1 Code Marking Overview

Code Protection allows you to manually insert specific `#pragma` statements, called code markers, directly into the source code to improve Code Protection knowledge about the source code structure and assumptions, as well as to explicitly tell Code Protection what level of security should be applied to individual functions or sets of functions. Although code marking is optional, it can significantly improve speed and security of the protected application.

Code marker `#pragma` statements must be inserted into the source code before the functions you want to mark.

7.2 Avoiding Compiler Errors

Since your compiler may display warnings or errors about unknown `#pragma` statements, we recommend that you surround Code Protection code markers with "`#ifdef __SCP__`" and "`#endif`" directives, for example as follows:

```
{ ... }
#ifdef __SCP__
#pragma NW group speed
#endif
{ ... }
```

In this way, code markers will be visible to Code Protection but not to standard compilers. For information on the "`__SCP__`" macro, see §1.2.8.

7.3 Grouping Functions

Better speed and security results can be achieved if Code Protection knows how application functions are logically grouped together. The following are the main motivators why we strongly advise function grouping:

- Instead of applying profiling results uniformly to the entire application, Code Protection will take into consideration function groups and adjust the security level accordingly.
- Some Code Protection security settings can be individually adjusted for each function group as described in §5.3.2. Therefore you can raise the security level for some functions, and reduce the level for others.

To specify which individual functions form separate logical units, you can insert the following statement directly into the source code before the functions you want to mark:

```
#pragma NW group «name»
```

This statement tells Code Protection that all subsequent functions in the code form one logical group called *«name»*. In this way, you can define as many function groups as necessary.

To mark an end of a group, you can insert the following statement after the last function:

```
#pragma NW group default
```

The `default` group is the one where all unmarked functions belong by default.

Besides the function groups you define, the following predefined function groups are always available:

`default`

This group is used by all unmarked functions.

`speed`

This function group is configured so that all its functions remain unmodified. These functions will still be protected by integrity checks invoked from other functions.

> ⚠ By including a function in the `speed` group you are significantly reducing its security level. If the function contains sensitive code, it will be more vulnerable to hacker attacks.

`security`

Functions in this group are optimized for security, regardless of profiling results.

`obfuscation`

Functions in this group will only be obfuscated (see §1.2.3.2), but other security mechanisms (such as performing integrity, anti-debug, jailbreak, rooting, and method swizzling checks) will not be embedded in these functions.

The predefined function groups described above can be modified using the GUI as described in §5.3.2.

If you specify multiple groups with the same identifier, Code Protection will collect all such groups into one group. For example, in the following code fragment, functions `foo2`, `foo3`, and `foo5` will be collected into one function group named `license_check`:

```
void foo1() { ... }
#pragma NW group license_check
void foo2() { ... }
void foo3() { ... }
#pragma NW group default
void foo4() { ... }
#pragma NW group license_check
void foo5() { ... }
```

> ⚠ If you have added or changed function group markers in the source code, you have to repeat the source code analysis step, as well as rebuild the profiling application and repeat the profiling process. Also, you will probably need to make changes in the Code Protection project to update the list of function groups as described in §5.3.2.

7.4 Code Marker Stack

Code Protection supports pushing (saving) the current function group name to the stack and popping (restoring) it at a later point in code (see §7.3). Such functionality may facilitate management of code markers in complex code.

At the beginning of each `.c` and `.cpp` source file, the stack is cleared and the function group name is reset to `default`. The stack is preserved and usable in the included `.h` and `.hpp` files.

7.4.1 Stack Statements

The following special statements are provided to use the code marker stack:

`#pragma NW push`

Pushes the current function group name to the stack (but does not change it).

`#pragma NW pop`

Pops the last function group name from the stack.

If this statement is called and the code marker stack is empty (the pop operation is unpaired with the push operation), the function group name is set to `default`.

7.4.2 Example

In the following example, functions `foo1` and `foo3` belong to the function group named `license_check`, but the function `foo2` belongs to the `default` function group:

```
#pragma NW group license_check
void foo1() { ... }
#pragma NW push
#pragma NW group default
void foo2() { ... }
#pragma NW pop
void foo3() { ... }
```

8 Using the Command-Line Tool

This chapter describes how to use the Command-Line Tool.

8.1 Command-Line Tool Overview

Code Protection provides a utility named Command-Line Tool, which can be included in scripts and automated processes. This tool can execute and upgrade existing Code Protection projects, but it cannot be used for creating new projects or modifying existing ones. You must always provide an existing project to the Command-Line Tool.

8.2 Running the Command-Line Tool

The Command-Line Tool is an executable file named `scp-tool`, which is located in the Code Protection installation folder. On OS X, the `scp-tool` file is located in the `Code Protection.app\Contents\MacOS` folder.

To run the Command-Line Tool, execute the following command:

```
scp-tool «action»
```

where *«action»* is one of the following:

`--analyze «project».nwproj`

Builds the original source code to understand which files need to be protected. This operation should be performed every time the source code is modified.

This operation does the same as the **Analyze** button in the **Analysis** page (see §3.2).

`--build-profiling «project».nwproj`

Builds the profiling application.

This operation does the same as the **Build** button in the **Profiling** page (see §4.2).

`--profile «project».nwproj [--continue]`

Starts the profiling process.

This operation does the same as the **Start Profiling** button in the **Profiling** page (see §4.2).

To add new profiling data to the existing data, append the parameter `--continue` to the command. This is the same as selecting the **Continue profiling** check box in the **Profiling** page.

`--load-profiling «project».nwproj «file».nwprofile [--continue]`

Loads the profiling statistics file (`*.nwprofile`) provided as a command line argument.

This operation does the same as the **Select File** button in the **Profiling** page (see §4.2).

To add new profiling data to the existing data, append the parameter `--continue` to the command. This is the same as selecting the **Continue profiling** check box in the **Profiling** page.

`--profiling-statistics «project».nwproj`

Displays in the console output how many times each function was called during the last profiling run.

`--profiling-csv «project».nwproj [«output».csv]`

Same as `--profiling-statistics`, but stores the report in a CSV file, which you can then open in any spreadsheet editor. The generated CSV file will be placed next to the Code Protection project.

This operation does the same as the **Save Results** button in the **Profiling Results** window (see §4.3).

If the CSV file name is not provided, it will be named "`«project name».csv`".

`--protect «project».nwproj`

Builds the final protected application.

This operation does the same as the **Protect** button in the **Protection** page (see §5.2).

`--protect-apk «project».nwproj «file».apk [«output name»]`

Creates the finalized copy of the specified protected APK package.

This operation does the same as the **Finalize APK Package** button in the **Protection** page (see §12.2).

Optionally, you can specify the name of the finalized APK file by providing it as a command-line argument. If the name is not specified, the finalized file will be named "`«original name»-finalized.apk`". For example, if the original APK file is named "`MyApp.apk`", the finalized file will be named "`MyApp-finalized.apk`".

For detailed instructions on APK package integrity protection, see §12.

`--protection-statistics «project».nwproj`

Displays in the console output a detailed report on functions in the protected application and individual security features applied to each of them.

`--protection-csv «project».nwproj [«output».csv]`

Same as `--protection-statistics`, but stores the report in a CSV file, which you can then open in any spreadsheet editor. The generated CSV file will be placed next to the Code Protection project.

This operation does the same as the **Save Results** button in the **Functions** tab of the **Protection Results** window (see §5.4.2).

If the CSV file name is not provided, it will be named "`«project name».csv`".

`--export-protection «project».nwproj «file».nwdb`

Creates a protection data file (see §6.1.2).

This operation does the same as the **Export Protection Data** button in the **Protection** page (see §5.2).

This file is only required if you are building a protected static library. The obtained protection data file must always be delivered together with the static library.

For information on working with protected static libraries, see §6.

`--upgrade «project».nwproj`

Upgrades the Code Protection project and associated databases to the current version as described in §2.3.4.

`--pack «project».nwproj «file»`

Attempts to apply binary packing (see §1.2.3.12) to the executable file or shared library that is specified by the argument that follows this command.

> ⚠ This action should only be used for packing third-party executables for which you do not have the source code. If you have the source code and you are protecting it using Code Protection, packing will be applied automatically during the protection stage (if packing is enabled in project settings).

`--help`

Displays command-line help.

`--version`

Displays version information.

9 Verifying Function Caller Modules

This chapter describes what verification of function caller modules is and how it can be enabled in Code Protection.

9.1 Overview of Function Caller Module Verification

In Code Protection, you can introduce some protection of sensitive functions against callers in unauthorized modules. This means that the protected application will detect caller modules that are not allowed to call sensitive functions, and defend itself if such modules are detected. For general information on this feature, see §1.2.3.10.

Here are the main steps that you need to take to use verification of function caller modules:

1. In the application source code, identify the sensitive functions that should be protected against unauthorized caller modules.
2. Somewhere in the body of these functions, add a call of a specific checker function, which determines the caller module of this function, and the corresponding action to be taken if the module is unauthorized (see §9.2).
3. When building the protected application, provide a list of modules that are allowed to call the protected functions (see §9.3).

⚠ We strongly recommend using this feature together with cross-checking of shared libraries, which is described in §10.

An example project demonstrating verification of function caller modules is included in the `examples.zip` file (see §1.4).

9.2 Adding Checkers

To introduce protection against unauthorized calling of functions, take the following steps:

1. Add the following directive to the beginning of the source file:

   ```
   #include "scp_check_dll_signature.h"
   ```

   To avoid compiler errors when building the application without Code Protection, we recommend to complement the `#include` directive as follows:

   ```
   #ifdef __SCP__
   #include "scp_check_dll_signature.h"
   #else
   #define SCP_VerifyCaller() (1)
   #endif
   ```

   For information on the “`__SCP__`” macro, see §1.2.8.

2. Add caller verification function calls as necessary.

   Verification is performed by a special function named `SCP_VerifyCaller`, which is declared as follows:

   ```
   int SCP_VerifyCaller();
   ```

   `SCP_VerifyCaller` is a predefined Code Protection function that returns 1 if the caller module of that particular function is authorized, and 0 if it is not.

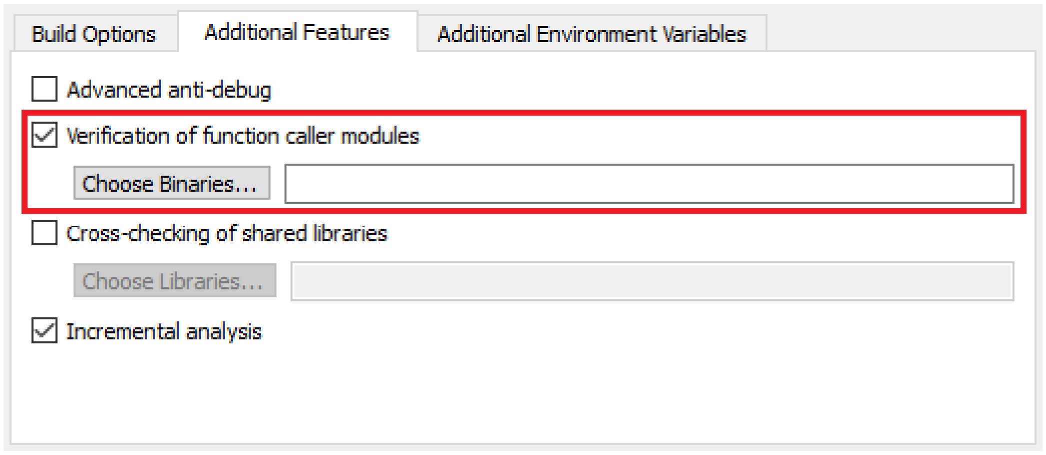
3. Since the `SCP_VerifyCaller` function does not do anything else except return 1 or 0, add corresponding logic to your code to respond to the result, such as terminate execution of the application if an unauthorized caller module is detected.

9.3 Specifying the Authorized Caller Modules

In addition to modifying the source code, you must also provide the white list of authorized modules to Code Protection. Signatures of these modules will be embedded in the protected application and used for verifying caller modules.

To specify the authorized caller modules, proceed as follows:

1. Open the corresponding Code Protection project as described in §2.3.2.
2. In the **Analysis** page, in the **Additional Features** tab, select the **Verification of function caller modules** check box as shown in Figure 9.1.



☐ Advanced anti-debug
☑ Verification of function caller modules
☐ Cross-checking of shared libraries
☑ Incremental analysis

Figure 9.1: Specifying authorized callers

3. Click **Choose Binaries**.

   The **Choose Binaries** window appears listing all binaries detected in your application's source root folder (see Figure 9.2).

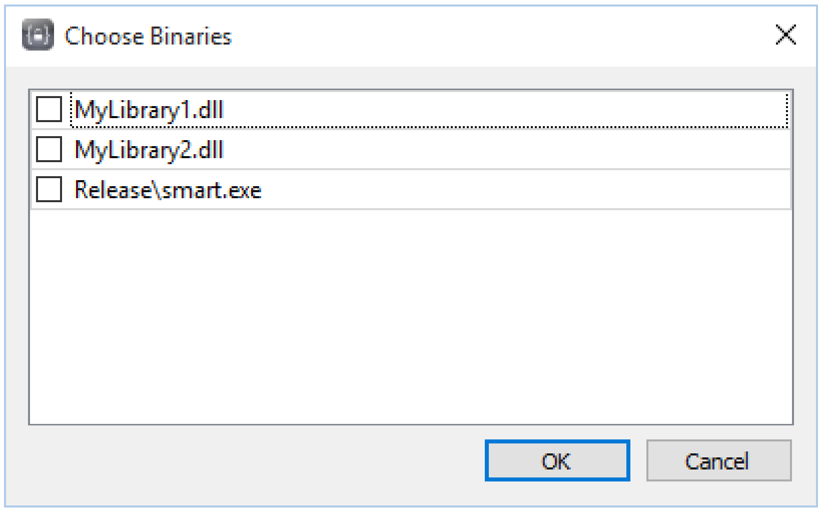


Figure 9.2: Detected binaries in the source root folder

You cannot select binaries that are located outside the source root folder.

4. Select check boxes for those binaries that are allowed to call the protected functions.
5. Click **OK**.
   The selected modules are displayed in the text field next to the **Choose Binaries** button.
6. Execute the source code analysis step as described in §3.
7. Build the final protected application as described in §5.

9.4 Example Project

The Code Protection package contains an example project that demonstrates function caller module verification on Windows. The example provides source code of a Windows app located in the `WindowsCallerVerifier` folder, which is compressed inside the `examples.zip` file.

The `WindowsCallerVerifier` folder includes a text file named `Instructions.txt`, which provides technical details about the example.

10 Cross-Checking of Shared Libraries

This chapter describes what cross-checking of shared libraries is and how it can be enabled in Code Protection.

10.1 Overview of Shared Library Cross-Checking

Code Protection can verify the integrity of shared libraries (`*.dll` or `*.so` files) that you deliver with your application. For more information on this feature, see §1.2.3.11.

> ⚠ The shared library files to be cross-checked must not be packed. It is expected that these are third-party files for which you do not have the source code and which cannot be protected with Code Protection.

An example project demonstrating cross-checking of shared libraries is included in the `examples.zip` file (see §1.4).

10.2 Enabling Cross-Checking of Shared Libraries

The following are the main steps that you need to take to use cross-checking of shared libraries:

1. In the application source code, choose the places where you want to perform cross-checking of shared libraries.

   > ⚠ Since cross-checking of shared libraries is a relatively expensive operation, you should not perform these checks in speed-sensitive functions.

2. In these places, add a call to a special checker function and specify the name of the particular shared library file to be verified (see §10.2.1).

   > ⚠ The shared library must be loaded into the memory before performing the check.

3. When building the protected application, enable cross-checking and specify the shared library files that will be cross-checked at run time (see §10.2.2).

> ⚠ For Windows applications, we recommend using this feature together with verification of function caller modules (see §9) for better security.

10.2.1 Adding Checkers

To add cross-checking of shared libraries to the protected application, in the chosen places of the source code, do the following steps:

1. Add the following directive to the beginning of the source file:

```
#include "scp_check_dll_signature.h"
```

To avoid compiler errors when building the application without Code Protection, we recommend to complement the `#include` directive as follows:

```
#ifdef __SCP__
#include "scp_check_dll_signature.h"
#else
#define SCP_CheckDllSignature(lib) (1)
#endif
```

For information on the "`__SCP__`" macro, see §1.2.8.

2. Add cross-checker function calls as necessary.

   Cross-checking is performed by a special function named `SCP_CheckDllSignature`, which is declared as follows:

   ```
   int SCP_CheckDllSignature(const char* name);
   ```

   where `name` is the shared library file name to be verified, for example "library.dll".

   The function returns 0 if the shared library is not loaded or if it has been modified, and 1 if the shared library is loaded and valid.

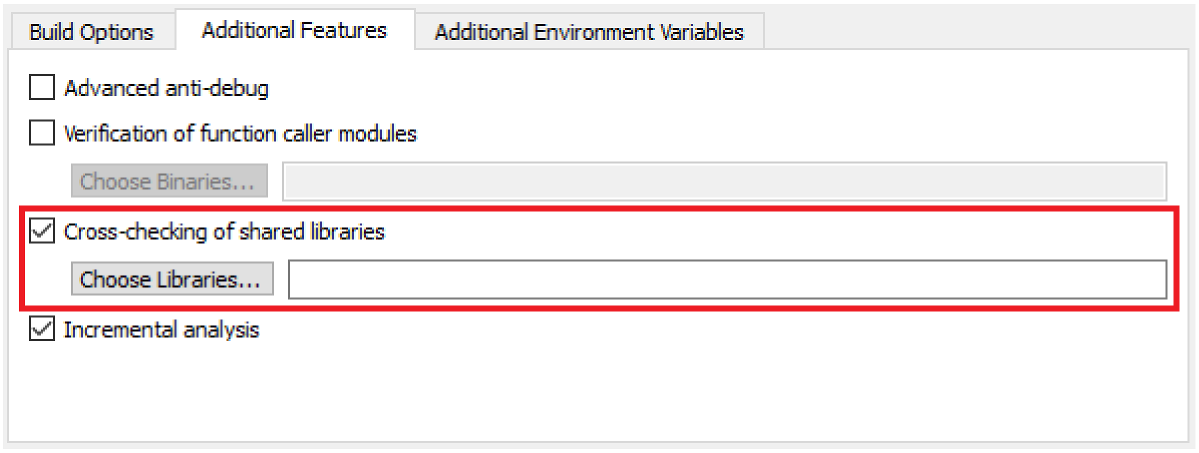
3. Since the `SCP_CheckDllSignature` function does not do anything else except return 1 or 0, add corresponding logic to your code to respond to the result, such as terminate execution of the application if a loaded shared library is modified.

10.2.2 Enabling Cross-Checking in the Project

In addition to modifying the source code, you must also specify which shared library files need to be subject to cross-checking. Signatures of these files will be embedded in the protected application and used for cross-checking at run time.

To enable cross-checking and specify the shared library files, proceed as follows:

1. Open the corresponding Code Protection project as described in §2.3.2.
2. In the **Analysis** page, in the **Additional Features** tab, select the **Cross-checking of shared libraries** check box as shown in Figure 10.1.



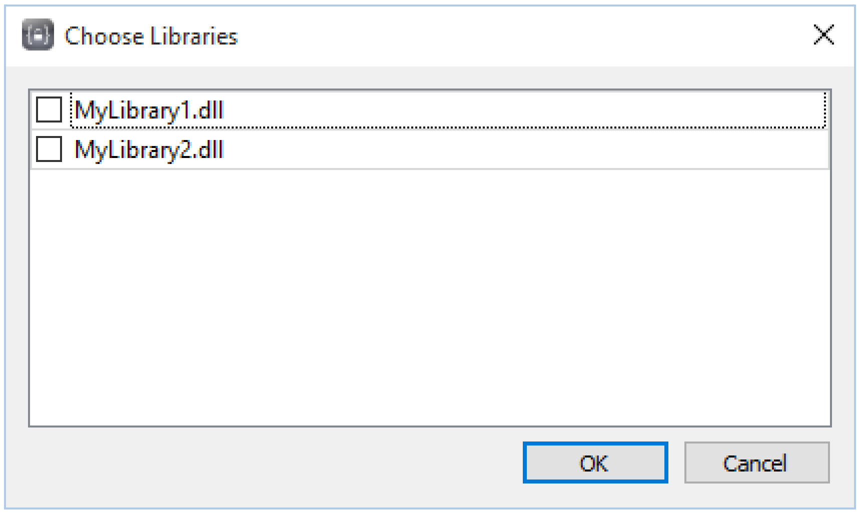
Build Options | Additional Features | Additional Environment Variables

☐ Advanced anti-debug
☐ Verification of function caller modules
Choose Binaries...
☑ Cross-checking of shared libraries
Choose Libraries...
☑ Incremental analysis

Figure 10.1: *Enabling cross-checking of shared libraries*

3. Click **Choose Libraries**.

   A window appears listing all the detected shared library files under the source root of the target application as shown in Figure 10.2. You cannot include files that are located elsewhere.



Choose Libraries

☐ MyLibrary1.dll
☐ MyLibrary2.dll

OK | Cancel

Figure 10.2: *Selecting the shared library files*

4. Select the necessary files and click **OK**.

   The selected files are displayed in the text box next to the **Choose Libraries** button. This text box cannot be modified directly.

5. Execute the source code analysis step as described in §3.

6. Build the final protected application as described in §5.

10.3 Example Projects

The Code Protection package contains example projects that demonstrate cross-checking of shared libraries on Android and Windows. The examples are located in the following folders:

`AndroidCrossSoCheck`
: contains an implementation for the Android platform

`WindowsDllCrossChecking`
: contains an implementation for the Windows platform

These folders are compressed inside the `examples.zip` file. Each folder includes a text file named `Instructions.txt`, which provides technical details about the corresponding example.

11 Securing the Google Play Licensing Service

This chapter describes how to increase the security of the Google Play licensing service within your application using Code Protection.

11.1 Google Play Licensing Overview

Google offers a licensing service that lets developers enforce licensing policies for Android applications published via Google Play. With Google Play licensing, an application can query Google Play at run time to obtain the licensing status for the current user, and then allow or disallow further use as appropriate. Essentially, this mechanism ensures that Android apps are not copied and distributed without the publisher's permission.

All the licensing-related communication is handled by Licensing Verification Library (LVL), a library provided by Google that you can include in your Android apps. LVL communicates with the Google Play client running on all Android devices, which in turn communicates with the Google Play server. This is shown in Figure 11.1.

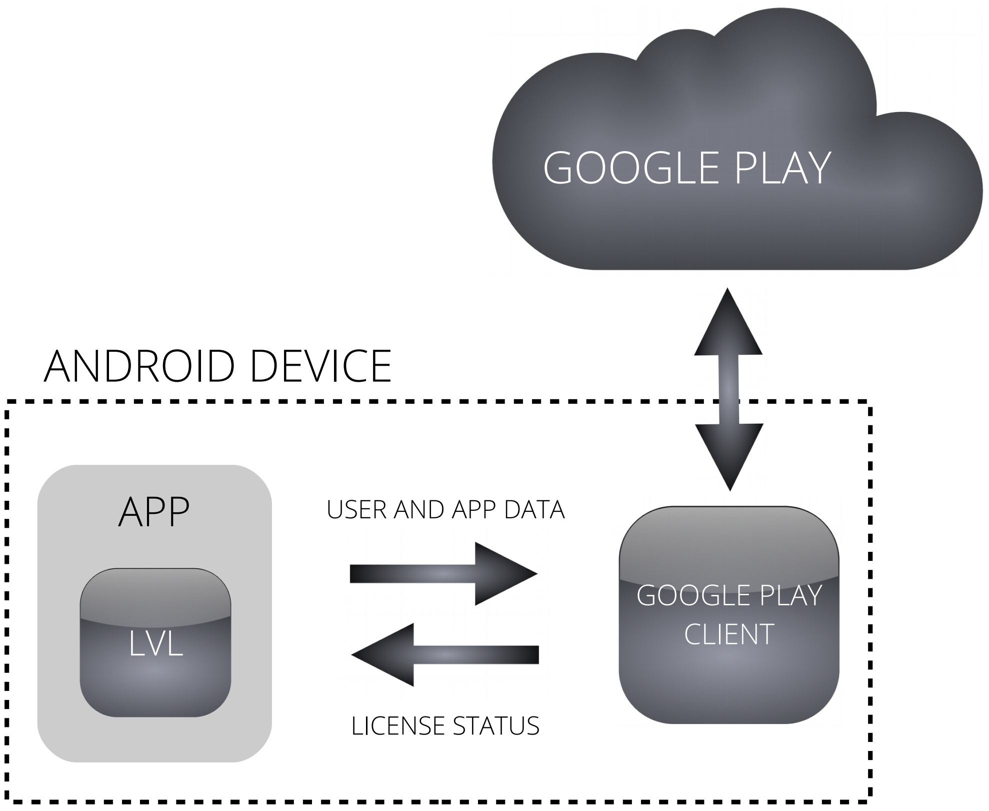


Figure 11.1: Google Play licensing scheme

For detailed information on Google Play licensing, see the following page:

developer.android.com/google/play/licensing/index.html

11.1.1 Threat

The standard implementation of LVL offered by Google is written in Java, which is highly vulnerable to reverse engineering and tampering. Because of this, a hacker can easily remove LVL from an app, and distribute the application to other users for free. Furthermore, Google's implementation of LVL is not compatible with many Java obfuscators, and thus many apps ship with unobfuscated copies of LVL. This makes it easy for hackers to use simple search-and-replace tools to automatically find and remove LVL from apps.

11.1.2 Protection

Code Protection fights against both automatic and manual cracking of LVL by using an alternative implementation of LVL entirely written in C, which exposes a simple API that your application can use to implement license checks. The implementation is kept secure by obfuscation, integrity checks, and anti-debug. Hackers will no longer be able to remove LVL from an application by decompiling and patching the Java binary. In order for a hacker to crack the native implementation of LVL, he would have to bypass all other security features of Code Protection, which is an extremely difficult task. Thus Code Protection will make it more difficult for your Android application to be cracked automatically, in a way that should thwart even skilled attackers.

11.2 Using the API

To use the API of the protected LVL implementation, you need to include the `scp_licensing.h` file in your native source code. This file is available in the "`additional/android`" subfolder. You do not have to copy this file into your project, or compile and link it. Code Protection will do it all for you.

The build system will recognize the `scp_licensing.h` file only if you have selected the **Google Play licensing protection** check box in the **Analysis** page as described in §3.2.

11.2.1 Functions

The `scp_licensing.h` file exposes four simple functions described in the following subsections. If a function is successfully executed, it returns 0. Any other return value means that there has been a problem with the function call.

11.2.1.1 SCP_License_Create

This is the first function that you call. It creates an `SCP_License` object, which you must then provide to all other API function calls. Since creating the `SCP_License` object is a relatively expensive operation, you should perform it only once, and release the object only when you are done with license verification.

The function is declared as follows:

```
int SCP_License_Create(const SCP_LicenseParams* params, SCP_License** lic);
```

The following parameters are used:

`params`

Pointer to the `SCP_LicenseParams` structure (see §11.2.2.2), which provides all the necessary input parameters.

You have to create this structure before calling the function.

lic

Address of a pointer to the `SCP_License` object that will be created by this function.

11.2.1.2 SCP_License_Release

This function releases the `SCP_License` object. Call this function only when all license verification checks are completed.

The function is declared as follows:

```
int SCP_License_Release(SCP_License* lic);
```

`lic` is a pointer to the `SCP_License` object that you previously created using the `SCP_License_Create` function.

11.2.1.3 SCP_License_Verify

This function is the one that actually initiates license verification via the Google Play client. You cannot call this function if a previous license verification process is not yet completed.

The function is declared as follows:

```
int SCP_License_Verify(SCP_License* lic, SCP_LicenseCallback callback, void* userData);
```

The following parameters are used:

lic

Pointer to the `SCP_License` object that you previously created using the `SCP_License_Create` function.

callback

Callback function that LVL invokes when the license check is completed. For details on this function, see §11.2.2.1.

userData

Optional pointer to an arbitrary data structure that you may want to pass to the callback function.

11.2.1.4 SCP_License_WaitForResult

If you call this function, execution of the calling thread will be paused until the license check is completed. This function will not return until the callback function has finished its execution.

The function is declared as follows:

```
int SCP_License_WaitForResult(SCP_License* lic);
```

`lic` is a pointer to the `SCP_License` object that you previously created using the `SCP_License_Create` function.

11.2.2 Supporting Data Structures

This section describes the additional data structures used by the API functions.

11.2.2.1 SCP_LicenseCallback

This is the function pointer type defining the callback function invoked by LVL when a license verification check is completed.

The function is declared as follows:

```
typedef void (*SCP_LicenseCallback)(
    void* userData,
    SCP_LicenseResult result,
    const unsigned char* userId,
    int userIdLength,
    const SCP_LicenseResultExtras* extras);
```

The following parameters are used:

`userData`

Optional pointer to an arbitrary data structure that you previously passed to the `SCP_License_Verify` function (see §11.2.1.3).

`result`

`SCP_LicenseResult` enumeration that will contain the license check result (see §11.2.2.3).

`userId`

Pointer to a byte array that will contain the unique user identifier.

`userIdLength`

Length of the user identifier in bytes.

`extras`

Pointer to the `SCP_LicenseResultExtras` structure that will contain additional data returned by the licensing server (see §11.2.2.4).

11.2.2.2 SCP_LicenseParams

This structure provides the main input parameters that you must pass to the `SCP_License_Create` function (see §11.2.1.1).

The structure is declared as follows:

```
typedef struct {
    JNIEnv* env;
    jobject context;
    const char* publicKey;
    unsigned int timeout;
} SCP_LicenseParams;
```

The following parameters are used:

`env`
: Pointer to the `JNIEnv` object.

`context`
: Subclass of the `android.content.Context` class, for example an instance of `android.app.Activity`.

`publicKey`
: Base64-encoded public key of your Google Play publisher account.

  You can get your public key here:

  https://play.google.com/apps/publish

`timeout`
: License verification timeout in seconds.

  0 means the default timeout, which is 10 seconds.

11.2.2.3 SCP_LicenseResult

This enumeration defines all the possible results of a license verification check. The following values are defined:

`SCP_RESULT_TIMEOUT`
: License verification did not end in the allocated time.

`SCP_RESULT_INVALID_SIGNATURE`
: The response from the licensing server contains an invalid signature.

`SCP_RESULT_INVALID_FORMAT`
: The response from the licensing server was presented in an invalid format.

`SCP_RESULT_LICENSED`
: The application is licensed to the user. The user has purchased the application or the application only exists as a draft.

`SCP_RESULT_NOT_LICENSED`
: The application is not licensed to the user.

`SCP_RESULT_LICENSED_OLD_KEY`
: The application is licensed to the user, but there is an updated application version available that is signed with a different key.

`SCP_RESULT_ERROR_NOT_MARKET_MANAGED`
: The application (package name) was not recognized by Google Play.

`SCP_RESULT_ERROR_SERVER_FAILURE`

The server could not load the publisher account's key pair for licensing.

`SCP_RESULT_ERROR_OVER_QUOTA`

The Google Play Android Developer API has reached the limit of queries per day.

`SCP_RESULT_ERROR_CONTACTING_SERVER`

The Google Play client was not able to reach the licensing server, possibly because of network availability problems.

`SCP_RESULT_ERROR_INVALID_PACKAGE_NAME`

The application requested a license check for a package that is not installed on the device.

`SCP_RESULT_ERROR_NON_MATCHING_UID`

The application requested a license check for a package whose UID (package-user ID pair) does not match that of the requesting application.

11.2.2.4 SCP_LicenseResultExtras

This structure contains additional data returned by the Google Play licensing server via the callback function (see §11.2.2.1).

The structure is defined as follows:

```
typedef struct {
    unsigned long long VT;
    unsigned long long GT;
    unsigned long long GR;
    char* fileUrl1;
    char* fileUrl2;
    char* fileName1;
    char* fileName2;
    unsigned long long fileSize1;
    unsigned long long fileSize2;
} SCP_LicenseResultExtras;
```

The parameters correspond to licensing server response data described here:

developer.android.com/google/play/licensing/licensing-reference.html

11.3 Limitations

When employing the native implementation of LVL, please take into consideration the following limitations:

- You should not implement the Google Play server result processing in Java (as it is done in the example app), because a skilled hacker can simply reverse engineer the Java code, and modify the result value returned by the callback function. All the sensitive operations related to license verification should always be implemented in native code, which is covered by Code Protection.

- Due to limitations of the Android system, if the API functions are invoked from the main application thread, they can freeze the application if the message queue of the main thread is not processed between the API calls. For this reason, we recommend that you always use the API from another thread.

11.4 Example Project

The Code Protection package contains an example project that demonstrates the usage of the native implementation of LVL. The example provides source code of an Android app located in the `AndroidLicenseCheck` folder, which is compressed inside the `examples.zip` file.

The `AndroidLicenseCheck` folder includes a text file named `Instructions.txt`, which provides technical details about the example.

12 Protecting Android Application Packages

This chapter describes how to use Code Protection to protect Android application packages against modification.

12.1 Overview of Android Application Packages

An Android application package (APK) is a file of a specific format that is used to distribute and install Android apps. It contains the entire application code and resources, as well as a digital signature created with the developer's private key.

A very common practice is that hackers modify an APK package and re-sign it with a different key, which in essence allows the application to be distributed freely without developer's consent.

Code Protection provides a set of security features, which enable you to harden the APK package against tampering. You can manually insert APK package integrity checks at arbitrary places in your code, which, at run-time, will check if the APK file is modified.

An example project demonstrating protection of Android application packages is included in the `examples.zip` file (see §1.4).

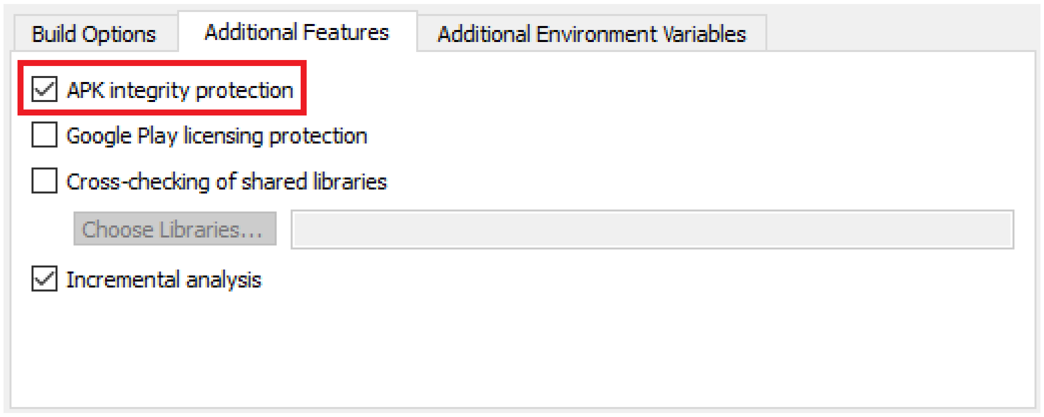
12.2 Enabling APK Package Protection

The following are the main steps you need to perform to apply APK package integrity protection:

1. In the application source code, choose the places where you want to insert APK package integrity checks.

   > ⚠ Since verification of an APK package is a relatively expensive operation, you should not perform these checks in speed-sensitive functions.

2. In the chosen places, add a call to a special checker function as described in §12.3.
3. Open the Code Protection project as described in §2.3.2.
4. In the **Analysis** page, select the **APK integrity protection** check box (see Figure 12.1).



- [✓] APK integrity protection
- [ ] Google Play licensing protection
- [ ] Cross-checking of shared libraries
- [✓] Incremental analysis

Figure 12.1: Enabling APK package protection in the project

This operation will allow the `scp_apk_integrity.h` header to be used in your native code.

5. In Code Protection, execute the analysis, profiling, and protection steps as usual (see §1.2.2).
6. After the protection step if finished, build the protected edition of your application's APK file using your build tool (such as Gradle, Eclipse, or Ant) and then sign the APK file.

   ⚠ The APK file must include native libraries from the protection folder (specified in the **Protected application folder** field), which was created by Code Protection during the protection step (see §5.2).

7. In the **Protection** page, click **Finalize APK Package** and select the APK file you created in step 6 to prepare the final protected APK package.

   The finalized copy of the APK file will be created and placed next to the original APK file (the one you created in step 6). The finalized APK file will be named "*`«original name»`*`-finalized.apk`". For example, if the original APK file is named "`MyApp.apk`", the finalized file will be named "`MyApp-finalized.apk`". The finalized APK file is the one that you should deliver to your customers.

   ⚠ Do not optimize the APK package with the `zipalign` tool as suggested by Google, because it will break the package integrity checksums.

Note that you can also use the Command-Line Tool to create the finalized version of the APK package as described in §8.2.

12.3 Adding Checkers

An APK package integrity check is performed by calling a special function named `SCP_CheckApkIntegrity`, which is declared as follows:

```
int SCP_CheckApkIntegrity(JNIEnv* env, jobject context);
```

The following parameters are used:

`env`

Pointer to the `JNIEnv` object.

`context`

Subclass of the `android.content.Context` class, for example an instance of `android.app.Activity`.

The function returns 0 if the APK package is intact and a different value if it has been tampered with.

Code Protection does not insert calls to this function automatically because performing an integrity check may take significant time and thus reduce your app's execution speed. You have to decide where to put these checks yourself.

To take advantage of the `SCP_CheckApkIntegrity` function, proceed as follows:

1. Include the header `scp_apk_integrity.h` in your native source code.

   Since your application may be compiled outside of Code Protection, we recommend that you include the header as follows:

```
#ifdef __SCP__
#include <scp_apk_integrity.h>
#else
int SCP_CheckApkIntegrity(JNIEnv* env, jobject context) {return 1;}
#endif
```

In this case, if your application is built without Code Protection, integrity checks will fail, but you will avoid compiler errors. For information on the "`__SCP__`" macro, see §1.2.8.

2. Where appropriate, include calls to the `SCP_CheckApkIntegrity` function and add corresponding logic to your code to respond to integrity check results, such as terminate execution of your application if APK package integrity is broken.

12.4 Example Project

The Code Protection package contains an example project that demonstrates the APK package protection feature. The example provides source code of an Android app located in the `AndroidApkIntegrityCheck` folder, which is compressed inside the `examples.zip` file.

The `AndroidApkIntegrityCheck` folder includes a text file named `Instructions.txt`, which provides technical details about the example.

13 Using Mach-O Binary Signature Verification

This chapter describes how you can protect iOS and OS X apps from unwarranted re-signing, which can be misused to facilitate application piracy as described in §1.2.3.7.

13.1 Protection Overview

Mach-O is an executable binary file format used primarily on iOS and OS X platforms. The Mach-O file format allows a signature to be attached to the file, which can be used to verify the distributor of the application. The device where an application is run uses public keys of known distributors (e.g. Apple or the application developer) to verify whether the file is unmodified and coming from a valid source.

The signature in the Mach-O file can be simply removed and replaced with another signature, generated by a different private key. This is a feature that is abused by hackers to distribute paid apps free of charge. The weakness lies in the fact that re-signing can be done any time without having access to the application's source code.

Code Protection can increase protection of iOS and OS X apps against unwarranted re-signing by embedding two public keys into the protected application during the protection stage:

- public key from Apple used for verifying apps downloaded from the App Store
- public key of the application developer

You can then add a special function call to the source code of the protected application to verify if the attached signature's public key matches either of the two public keys. In other words, this function will make sure that the application is either obtained from the official App Store (is legally purchased by the user) or is being used by its real developer (is in the development stage on an approved development device). The two embedded public keys are protected against modification, which means that if a hacker re-signs the app, neither public key will match the signature, and the application will know that it needs to defend itself.

An example project demonstrating Mach-O binary signature verification is included in the `examples.zip` file (see §1.4).

13.2 Enabling Mach-O Binary Signature Verification

This section describes the steps you need to perform to apply Mach-O signature protection.

1. Decide in which parts of your source code you will insert the function call that checks the signature.

   The signature check is performed by calling a special function `SCP_CheckSignature()`.The function will return 1 if the signature's public key matches either the Apple public key or the public key of the original developer. A different return value will indicate that the application has been re-signed.

   Code Protection does not insert calls to this function automatically because performing a signature check takes significant time and thus reduces your app's execution speed. You have to decide where to put these checks yourself.

2. To take advantage of the `SCP_CheckSignature()` function, include the header `scp_signature.h` in your source code.

Since your application may be compiled outside of Code Protection, we recommend that you include the header as follows:

```
#ifdef __SCP__
#include <scp_signature.h>
#else
#define SCP_CheckSignature() (1)
#endif
```

For information on the "`__SCP__`" macro, see §1.2.8.

3. Where appropriate, include calls to the `SCP_CheckSignature()` function and add corresponding logic to your code to respond to the case when the application is re-signed.

4. In the GUI, open the **Analysis** page and in the **Additional Features** tab, select the **Mach-O binary signature verification** check box as described in §3.2.

   This operation will allow the `scp_signature.h` header to be used in your source code.

   If you are protecting a static or dynamic library, you may not know which applications will use your library and what the developers' public keys of those applications will be. For this case, an additional text field **Extract the public key from an existing app** is provided underneath the **Mach-O binary signature verification** check box. If you select an application in the text field, Code Protection will extract the public key from its signature and embed this key into the protected library. Then this key will be used by the `SCP_CheckSignature` function to verify the signature of the final application.

5. Execute the source code analysis step as described in §3.

6. Optionally, execute the profiling step as described in §4.

7. Build the final protected application as described in §5.

13.3 Example Project

The Code Protection package contains an example project that demonstrates Mach-O binary signature verification. The example provides source code of an iOS app located in the `MachOSignatureCheck` folder, which is compressed inside the `examples.zip` file.

The `MachOSignatureCheck` folder includes a text file named `Instructions.txt`, which provides technical details about the example.

14 Configuring Compiler Proxies

In some cases, the build system requires some manual configuration steps to be executed before protecting your applications with Code Protection. This chapter describes why, when, and how this should be done.

14.1 Overview of Compiler Proxies

Code Protection has several internal components called compiler proxies, which are files located in the Code Protection installation folder. These proxies work as mediators between Code Protection and various system compilers. For more information on how Code Protection interacts with the build system, see §1.2.4.

Code Protection has separate compiler proxies for different target platforms (for Windows, there is a proxy for the linker as well). In most cases, Code Protection is capable of setting up these proxies without user's intervention. However, in some specific situations, you have to set up the proxies manually, as described in the subsequent subsections.

14.2 Setting Up the Compiler Proxies for Linux

The compiler proxies for Linux are files named `scp-unix-gcc` and `scp-unix-g++`. Due to the diversity of Linux applications and build systems, currently Code Protection is not able to automatically set up these proxies for Linux, and you always have to take care of it manually.

The subsequent two sections describe two different procedures that you need to execute to set up the compiler proxies for your application, depending on the build system you use.

14.2.1 Configuring Autotools and CMake

Code Protection takes advantage of the fact that Autotools and CMake support the `CC` and `CXX` environment variables. You have to execute this procedure once before you start protecting your application with Code Protection. You can follow this procedure for other similar build systems that support the `CC` and `CXX` environment variables.

To set up compiler proxies for an application built with Autotools or CMake, in your application's configure step, add the `CC` and `CXX` environment variables pointing to the compiler proxies before the `./configure` or `cmake` command.

So, for Autotools, the configuration command would be the following:

```
$ CC="«path to scp-unix-gcc»" CXX="«path to scp-unix-g++»" ./configure «other arguments»
```

For CMake, the command would be the following:

```
$ CC="«path to scp-unix-gcc»" CXX="«path to scp-unix-g++»" cmake
-DCMAKE_USE_RELATIVE_PATHS=ON «other arguments»
```

You can exclude either the `CC` or `CXX` variable if the corresponding compiler is not required for your application.

⚠ Autotools and other build systems may execute compiler tests during the initial configuration. C or C++ source files generated during these tests (such as `conftest.c`) should be excluded from protection.

14.2.2 Configuring SCons

SCons does not support the `CC` and `CXX` environment variables. Therefore, Code Protection achieves similar results using the `PATH` environment variable. You can follow this procedure for other similar build systems that do not support the `CC` and `CXX` environment variables.

To set up the Linux compiler proxies for an application built with SCons, proceed as follows:

1. Create a new folder anywhere on your workstation.

   You can name the folder any way you like.

2. In the created folder, create the following symlinks:

   - symlink named `gcc` pointing to the `scp-unix-gcc` file
   - symlink named `g++` pointing to the `scp-unix-g++` file
   - symlink named `gcc-original` pointing to the system's default C compiler
   - symlink named `g++-original` pointing to the system's default C++ compiler

   If you are only building C code, the `g++` and `g++-original` symlinks are not required. Similarly, if you are only building C++ code, the `gcc` and `gcc-original` symlinks are not required.

3. Every time you start Code Protection, modify the `PATH` variable so that the folder created in step 1 is processed before the system's default compiler, for example as follows:

   ```
   $ PATH="«path to the symlinks folder»:$PATH" ./scp
   ```

   This will force SCons to call the compiler proxies instead of the default compilers. The compiler proxies in turn will then invoke the standard compilers via the `gcc-original` and `g++-original` symlinks.

14.3 Setting Up Proxies for Custom Builds on Windows

For the Windows target, the following proxies are used by Code Protection:

- compiler proxy `scp-windows-cl.exe` that works as a mediator between Code Protection and the Visual Studio compiler `cl.exe`
- linker proxy `scp-windows-link.exe` that works as a mediator between Code Protection and the Visual Studio linker `link.exe`

If you use the default Code Protection build option for Windows, Code Protection automatically sets up these proxies. However, if you use the custom build option and it invokes `msbuild.exe`, you have to explicitly provide links to the proxies by adding the following two parameters to the build command:

- `/p:CLToolExe="%NW_CL%"`
- `/p:LinkToolExe="%NW_LINK%"`

`NW_CL` and `NW_LINK` are environment variables set by Code Protection that point to the corresponding proxies.

If you use the custom build option and it invokes `msbuild.exe`, we strongly recommend that you also add the following parameters to the build command:

- `/p:TrackFileAccess=false`
- `/p:LinkIncremental=false`
- `/t:Rebuild`

If you use the custom build option with a build system other than `msbuild.exe`, it is your task to make sure your build system invokes the `scp-windows-cl.exe` and `scp-windows-link.exe` proxies instead of `cl.exe` and `link.exe` files.

14.4 Setting Up Proxies for Custom Builds on Android

The compiler proxies for Android are files named `scp-android-gcc` and `scp-android-g++`. If you use the default Code Protection build option to build your Android applications, Code Protection automatically sets up these proxies. However, if you use the custom build option, and it invokes `ndk-build`, you have to provide two parameters to the build command as follows:

- If you use a Windows development workstation, provide the following parameters:
  - `TARGET_CC="%NW_TARGET_CC%"`
  - `TARGET_CXX="%NW_TARGET_CXX%"`
- If you use a Unix development workstation, provide the following parameters:
  - `TARGET_CC="${NW_TARGET_CC}"`
  - `TARGET_CXX="${NW_TARGET_CXX}"`

`NW_TARGET_CC` and `NW_TARGET_CXX` are environment variables set by Code Protection that point to the corresponding compiler proxies. If you are only building C code, the `TARGET_CXX` parameter can be excluded. Similarly, if you are only building C++ code, the `TARGET_CC` parameter can be excluded.

If you use the custom build option with a build system other than `ndk-build`, it is your task to make sure your build system invokes the `scp-android-gcc` and `scp-android-g++` proxies instead of invoking the compilers directly.

14.5 Setting Up the Compiler Proxy for Custom Builds on OS X and iOS

The compiler proxies for OS X and iOS are files named `scp-universal-clang` and `scp-universal-clang++`. If you use the default Code Protection build option to build your OS X or iOS applications, Code Protection automatically sets up these proxies. However, if you use the custom build option, and it invokes `xcodebuild`, you have to execute the following steps:

1. Add the following parameters to the build command:
   - `CC="${NW_CC}"`
   - `CXX="${NW_CXX}"`

- `LD="${NW_CC}"`
- `LDPLUSPLUS="${NW_CXX}"`
- `GCC_PRECOMPILE_PREFIX_HEADER=NO`
- `OBJROOT="«build folder»"`
- `SYMROOT="«build folder»"`

  where `NW_CC` and `NW_CXX` are environment variables set by Code Protection that point to the corresponding compiler proxies, and *`«build folder»`* is an arbitrary folder where you want the built files to be placed. The `GCC_PRECOMPILE_PREFIX_HEADER`, `OBJROOT`, and `SYMROOT` parameters must be set to prevent Xcode from generating hash values in file paths, which in turn create problems in the protection step.

  If you are only using Clang, the `CXX` and `LDPLUSPLUS` parameters can be excluded. Similarly, if you are only using Clang++, the `CC` and `LD` parameters can be excluded.

2. Make sure the Xcode setting **Derived Data** is set to **Relative**.

   To view and modify this setting, open Xcode and select **Preferences > Locations**.

If you use the custom build option with a build system other than `xcodebuild`, it is your task to make sure your build system invokes the `scp-universal-clang` and `scp-universal-clang++` proxies instead of invoking Clang and Clang++ directly.

15 Protecting Applications on Custom Platforms

This chapter describes how to use the custom platform option in Code Protection to protect applications for platforms that are not directly supported by Code Protection.

15.1 Custom Platform Overview

Code Protection does not fully support all target platforms used in the world, especially those that use proprietary toolchains. However, Code Protection can still be used to provide certain security features to applications built for unsupported platforms. This can be achieved by using the custom platform option. It is a combination of techniques that require you to write a plugin that implements the critical parts necessary for your target platform, which can then be used by Code Protection in the full protection cycle to produce a protected application.

While keeping in mind how Code Protection interacts with supported target platforms (see §1.2.4), consider how integration with a custom platform is established.

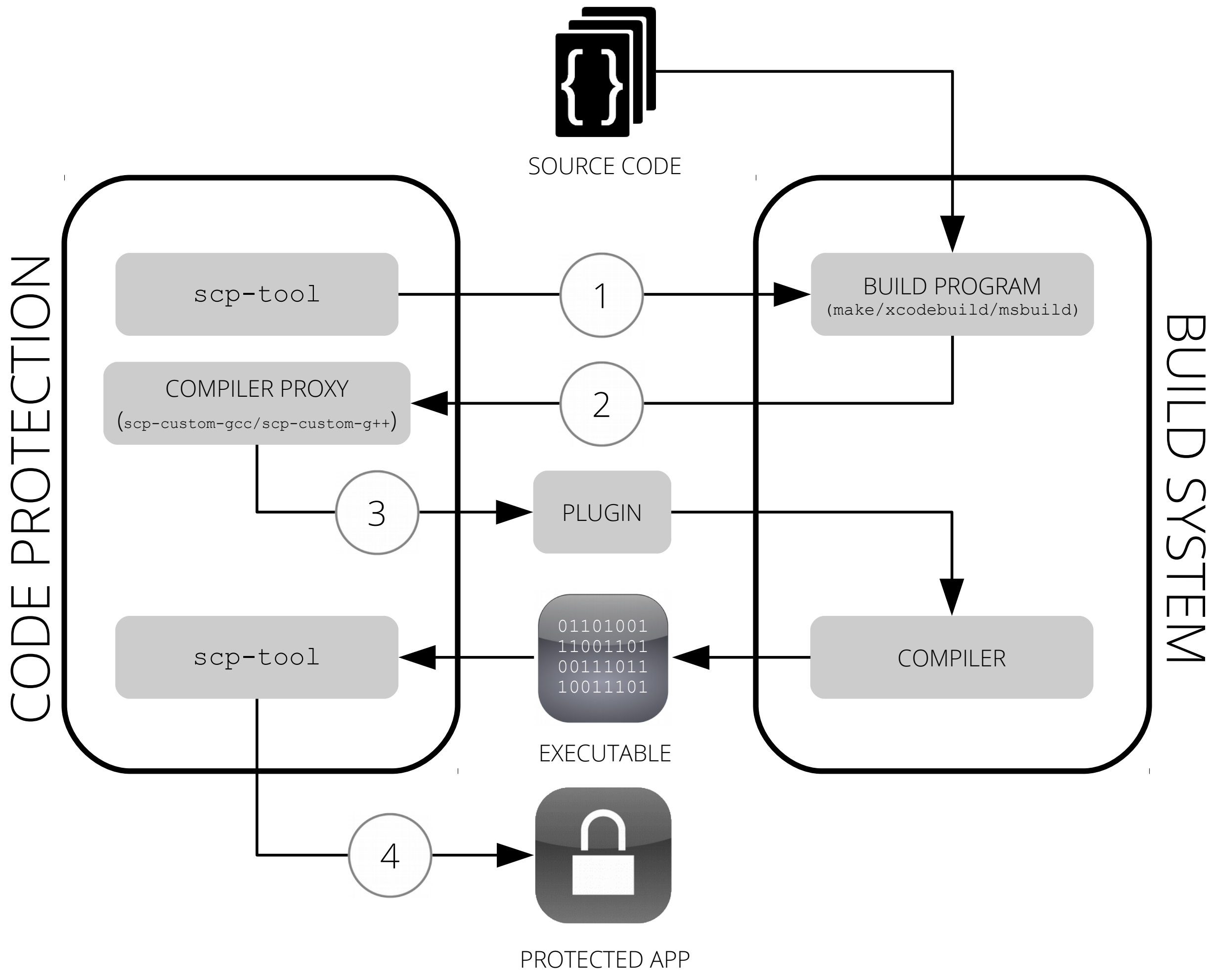


Figure 15.1: Integrating Code Protection with a custom platform

The following procedure explains the steps highlighted in Figure 15.1:

1. `scp-tool` invokes the system's build program via the custom build command option (see §3.2.2).

   The default build command is not available for custom platforms.

2. Code Protection instructs the build system to invoke the special compiler proxy for custom platforms (`scp-custom-gcc` or `scp-custom-g++` file depending on whether you are building C or C++ code).

   You have to configure your build system to use the compiler proxy similarly as described in §14.2.

3. The compiler proxy invokes the plugin, which is specified in the Code Protection project settings.

   This is the plugin that you have to write yourself and build as a shared library. It is the plugin's responsibility to correctly parse arguments, run the preprocessor, and invoke the original compiler (see §15.4).

4. In the protection stage, the compiled binary executable is again passed to `scp-tool` to update the embedded checksums for integrity protection (see §1.2.3.1).

15.2 Building a Protected Application for the Custom Platform

The following is the general procedure that you have to execute to build a protected application for a custom platform:

1. Write a plugin that implements the functionality required by Code Protection as described in §15.4.

2. Place the compiled plugin (shared library) in the root folder of the Code Protection package, next to the `scp-tool` executable.

   ⚠ The file name of the shared library must contain the word "plugin". Otherwise, Code Protection will not recognize it as a plugin.

3. Write a profiling function that creates and updates the offline profiling statistics file and add it to your project as described in §15.5.

4. Configure your build system to use the `scp-custom-gcc` and `scp-custom-g++` compiler proxies similarly as described in §14.2.

5. Create a new Code Protection project as described in §2.3.1 and make sure you select the **Custom platform** option.

6. Open the **Analysis** page and perform the following steps:

   - In the **Source root** field, select the root folder of the application's source code.
   - In the **Plugin** list box, select the shared library containing the plugin implementation.
   - In the **Custom build command** field, enter the command you use to build the application. The default build command is not available for custom platforms.

7. Click **Analyze**.

8. Build the profiling application and profile it as described in §4.

   The network profiling option is not available for custom platforms.

9. Configure security settings and build the protected application as described in §5.

   For custom platforms, only a subset of security features is available in the **Advanced Options** dialog (see §15.3).

15.3 Limitations

The custom platform option has the following limitations:

- You can use only the Linux development platform for building the target application.
- Your toolchain must use a GCC-based compiler.
- Only the following Code Protection features are supported:
  - integrity protection (see §1.2.3.1)
  - code obfuscation (see §1.2.3.2)
  - inlining of static void functions (see §1.2.3.13)
  - string literal obfuscation (see §1.2.3.16)
  - custom build command (see §3.2.2)
  - offline profiling (see §4.2.1)
  - excluding files and folders from processing (see §2.4)

15.4 Creating a Plugin for the Custom Platform

This section describes how you can create a plugin to be used by Code Protection for your custom platform.

15.4.1 Plugin Interface

To write a plugin, you have to provide an implementation for the `nw-plugin.h` interface, which is located in the root of the Code Protection package, and build it as a shared library.

⚠ The `nw-plugin.h` file is provided only in the Linux version of the Code Protection package.

The shared library should export only one function with the following prototype:

```
Toolchain1* SCP_CreateToolchain(uint32_t version, void* extra);
```

For detailed information on the plugin interface, read comments included in the file.

15.4.2 Example Plugin Implementation

The Linux version of the Code Protection package includes an example plugin implementation named `nw-plugin-linux.cpp` (located in the root folder of the Code Protection package) that is a simple implementation of the `nw-plugin.h` interface. This file contains extensive comments and can serve as a basis for your implementation.

The example implementation assumes the following statements are true about your platform and toolchain:

- Only ELF output binaries are supported.
- Only position independent code is built (no relocations in `.text` or `.rodata` sections).
- All executable code is in one `.text` section, and all read-only data is in the `.rodata` section.
- The compiler is called with the `-c` argument multiple times using one or more `.c` or `.cpp` files as input to get a preprocessed input source.
- The linker is called with the `-o` argument specifying the output file.
- All input source files on one command-line are written in the same language (C or C++) for each execution. Different executions can have different languages.

These limitations apply only to the example plugin, not to Code Protection in general. You can modify the example plugin to work with any other requirements.

To specify the GCC toolchain to be used, you have to set an environment variable named `NW_ORIGINAL_PREFIX`. The example implementation assumes you have `${NW_ORIGINAL_PREFIX}gcc` and `${NW_ORIGINAL_PREFIX}g++` executables. For example, if the GCC you want to use is `/opt/something/bin/mipsel-linux-uclibc-gcc`, then you need to set the value of the `NW_ORIGINAL_PREFIX` variable to "`/opt/something/bin/mipsel-linux-uclibc-`". If this environment variable is not set, the example implementation will use the default value "`/usr/bin/`", which is your host GCC.

For your convenience, a compiled version of the example plugin is provided in the file `libnwpluginlinux.so`, which is located in the root folder of the Code Protection package.

15.5 Supporting Profiling on Custom Platforms

In addition to writing the plugin, you must also write a special profiling function that will take care of creating and updating the offline profiling statistics file (see §4.2.1). This section describes what you need to know to implement this function.

15.5.1 Function Specification

Code Protection expects that you have the following function defined somewhere in your source code:

```
void __nw_ProfileUpdateFile(
    uint32_t    funcId,
    const char* profilingFilePath,
    int         deleteProfilingFile)
```

When called, this function must update the profiling file, specified in the `profilingFilePath` parameter, with information that the function whose ID is `funcId` was called one time.

The following are the rules that this function must adhere to:

- If the `deleteProfilingFile` parameter is not equal to 0, the function must delete the profiling file upon the first call.

- The `__nw_ProfileUpdateFile` function must be thread-safe, because it may be called simultaneously from multiple threads.
- The profiling file must correspond to the format shown in Figure 15.2.

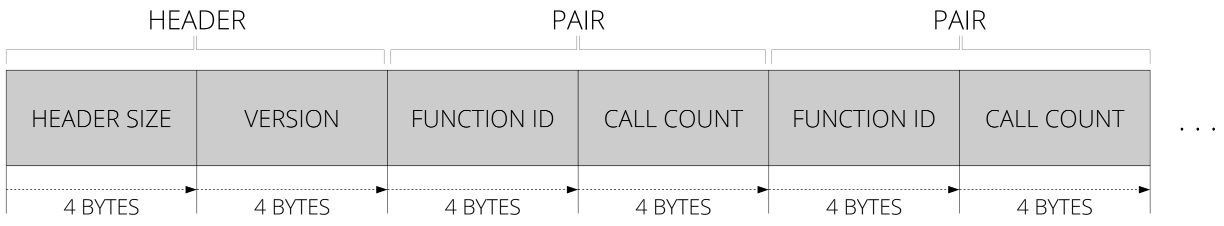


Figure 15.2: Profiling file format

  The first 8 bytes in the file contain the header, which consists of two 4-byte values. The first 4 bytes specify the header size, which must contain the value 8. The next 4 bytes specify the file version, which must contain the value 1.

  The header is followed by an unspecified number of pairs, consisting of two 4-byte values — the first 4 bytes contain the function ID, and the second 4 bytes contain the number of times the function was called. The same function ID can appear several times in the file in any order. The number of calls of each individual function will be added together from all function pairs.

- All values in the profiling file must be encoded in little-endian.

15.5.2 Example Implementation of the Profiling Function

The Code Protection package contains an example implementation of the profiling function. This implementation is available in the `profile-lib/profile-offline.c` file. You may use this implementation as is (should work for most cases), or use it as a basis for your custom implementation.

There are several ways how you can include the example implementation in your project:

- Copy its contents into your source code.

  > ⚠ In this case, you have to exclude the file containing this function from processing (see §2.4) to avoid an infinite loop when running the profiling application.

- Include the entire `profile-offline.c` file into your project by copying it into the source root.

  > ⚠ In this case, you have to exclude this file from processing (see §2.4) to avoid an infinite loop when running the profiling application.

- Include the file specified in the `NW_PROFILE_FILE` environment variable into your project. Code Protection always sets this variable to point to the original `profile-offline.c` file (the one in the `profile-lib` folder) when building the target application.

16 Troubleshooting

This chapter provides solutions to common problems that may occur with Code Protection.

➔ **The compiler crashes on a particular function or file.**

In this rare case you can try the following actions to resolve this problem:

- Try disabling protection for the function that is causing the crash. In the source code, add the "`#pragma NW group speed`" statement before the function as described in §7.3.
- If the above action does not help, try excluding the problematic source file from processing as described in §2.4.

➔ **During the analysis step, Code Protection reports an error about missing input files.**

This may happen if your build system is not configured to support Code Protection compiler proxies. Please follow instructions in §14 to resolve this problem.

➔ **During the protection step, Code Protection reports an error saying that a certain protection feature was not applied, followed by the sentence "Please verify your source code and/or adjust protection settings."**

This means that Code Protection cannot find any place where to insert integrity checks, anti-debug checks, anti-jailbreak checks, rooting checks, or method swizzling checks. Typically, this happens because the protected application contains too little code or too few functions.

Here is what you can try to do:

- Increase the protection value of the corresponding feature in the **Advanced Options** window as described in §5.3).
  For example, if this error is related to anti-debug checks, you would increase the **Anti-debug protection** slider value.
- Expand your source code.
- Disable the corresponding protection feature in the **Advanced Options** window.

➔ **You have built the profiling or protected version of a static library using Code Protection, but the final executable that uses this library fails to build.**

This may happen because you have not linked additional static libraries generated by Code Protection during the profiling or protection step. For more information, see §6.1.1.

➔ **Code Protection displays an error related to symlink creation while building the profiling application or the final protected application.**

This may happen if your build process creates symlinks while building the application. To work around this problem, you must delete these symlinks from the source folder before the profiling and protection stages are started.

➔ **The linker cannot link libraries when building the profiling or protected application for OS X or iOS. Errors about undefined symbols are displayed.**

This may happen only when you have Clang modules enabled in your Xcode project settings. To resolve this problem, try disabling the **Link Frameworks Automatically** option in Xcode.

➔ **Code Protection displays errors about temporary “ccache” files when building the profiling application or the final protected application for Android.**

Code Protection cannot protect Android applications if the `NDK_CCACHE` environment variable is defined. Make sure you have not defined this variable.

➔ **The profiling application cannot connect to Code Protection.**

Try the following steps to resolve this problem:

- Make sure you provided a correct IP address and TCP port of the computer where Code Protection is run when building the profiling application (see §4.2.2).
- Make sure the computer where Code Protection is run is reachable over the local network.
- Make sure the TCP port used for communication between Code Protection and the profiling application is not restricted.
- Make sure you have allowed the application to use network.
- If none of the above steps helped solve the problem, check the device log for any errors.

➔ **The protected application crashes at run time and debug logging shows that an integrity check is triggered.**

This may happen if the protected application is using a static library protected by Code Protection, but the protection data file (see §6.1.2) of the static library was not put next to the library file during the protection process or it had a different name.

Build the protected application again, but this time make sure that the protection data file is located next to the protected static library and has the same name (see §6.2).

➔ **An unprotected application that uses a static library protected by Code Protection crashes at run time and debug logging shows that an integrity check is triggered.**

This happens if you have not updated the application with the Binary Update Tool as described in §6.3.

➔ **The protected application is slow.**

Protected code will always be slower than unprotected code. This is a basic fact that cannot be completely avoided. However, you can significantly improve execution speed by applying the following techniques:

- Profile the application as described in §4.
  This should be the first option for improving execution speed.
- Adjust the protection slider as described in §5.2.
  By lowering the protection value, you will reduce security but also increase execution speed.
- Adjust settings in the **Advanced Options** window as described in §5.3.
- Organize source code functions into logical groups so that profiling statistics are applied relatively as described in §7.3.

➔ **The compiler reports errors or warnings about unknown `#pragma` statements.**

This may happen when the compiler encounters Code Protection code markers. To avoid this problem, make sure you follow the instructions in §7.2.

➔ **When Xcode is used, the compiler reports an error saying that the define set is not unique.**

This problem may occur when you have recursive header search paths enabled. In such case, you must make sure the include folder structure does not change between the analysis step and the profiling and protection steps. For example, the build folder must not be a subfolder in the header search paths.

➔ **I am experiencing a problem that is not listed here.**

Here is what you can do:

- Make sure your development and target environments are supported by Code Protection as described in §1.3.
- Carefully review the list of Code Protection limitations and known problems in §1.5.
- If you do not find a solution in the sections listed above, write to devsupport@whitecryption.com.