HIKOS - Highly Secure, Intelligent Software Copy-Protection

Fatbardh Veseli

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2011

Revision history

Version #	Description of change
0.1	Research Project Plan, first version, 13 December 2010.
0.2	Research Project Plan, second version (integrated the feedback suggestions), 21 December 2010.
0.7	A draft of the Master Thesis report is compiled and sent to my supervisor(s) and my student opponent by 31 may 2011.
1.0	The final Thesis Report, 1 July 2011.

1 Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

On the other hand, a part of my work is supervised under a Non-Disclosure Agreement, signed between *escrypt* and myself. Therefore, publishing sentitive parts of the project was restricted, as was the case with implementation details and own-created algorithms and source code for the HIKOS project. However, the main functionalities and results are described in the thesis so as to make the thesis readable and understandable for the audience.

Fatbardh Veseli

Abstract

Software piracy, which includes reproducing and distributing software products illegaly and without authorization, continues to cause financial losses to software vendors. Dongles for hardware protection have been present for a while now, but unfortunately, most of these solutions were only effective for a short period, until these methods were circumvented and time has showed that solutions that offer full protection from this phenomenon are impossible. In this project, we focus on a hardware copy protection scheme based on dongles and we take the current state-of-the-art in this area to a higher level. We provide a Highly Secure Software Copy Protection scheme and our contribution consists of the ability of the dongle to execute a selected part of the software inside the trusted environment in a dongle. This way, an attacker will never have the full software available on the host, which makes reverse-engineering of it more difficult.

Additionaly, I provide a threat model and security analysis for this mechanism. Finally, I attach parts of some algorithms I created for the Code Preprocessing, Analysis and Extraction.

Preface

Work on my thesis has been a very valuable experience for me, both on the professional and personal level. It was a pleasure to work under the supervision of Professor Paar and work in the warmly equipped environment at the Chair for Embedded Security under Ruhr University Bochum. On the other hand, I was always supported by my supervisor, Prof.Stephen Wolthusen, who has always given me helpful feedback and oriented me in the right path during this period. Also, the invaluable feedback and guidance from my thesis assistant, Dipl. Ralf Zimmerman, was a true asset during this period. Also, it was a pleasure to meet and cooperate with the wonderful team at *escrypt* and I would like to express my graditude for Annika Paus and Oliver Mischke for their feedback and cooperation. I am truly thankful and happy to have had the chance to cooperate with all of you.

On the other hand, when the motivation went down every once in a while, there was always some cheering up with my colleagues from our student-office. It was always refreshing to take a lunch break at the mensa with Pascal Beurer, have random coffees and refreshin talks with Roman Kochanek and everyone at the room 605. Also, I would like to thank Mrs. Irmgard Kuhen for all the help with the administrative and practical guides in Bochum. The same is true about the whole team at *emsec* for their nice welcome and the wonderful environment. The team has a full potential and energy, which I find amazing.

I should not forget to thank my friends Damla and Bylbyl for the wonderful dinners together when I came tired after the daily work. I truly appreciate the friendship and your warmth during the whole semester.

Finally, I wish to thank all the Erasmus students and great friends I made in Bochum, who made me forget the thesis troubles in the evenings and during our splendid trips in the week-ends, which made this a semester to remember.

Fatbardh Veseli, June 2011

Contents

List of Figures

List of Tables

2 Introduction

Living in an information age, it is noticable how much the computers are used and how important they are for many of our everyday tasks. Software solutions exist for solving many problems we encounter. Developing a specific software usually takes a lot of efforts and investment. The written lines of source code have a high production value, but no real material value.

Unfortunately, they are easy to copy and the developers (vendors) search for ways to protect this intellectual property from unauthorized (unpaid) use. Therefore, different licensing schemes were proposed to put control in the software in order to deny or grant the use of software [7].

2.1 Topic covered by the project

To software vendors, Intellectual Property is more than just an asset in the traditional business sense, but rather the product - it is the core business product and a main source of revenues (income). Thus, the need to protect this intellectual property becomes crucial.

This project focuses on creating a mechanism which offers a better software protection by using a hardware dongle. A dongle itself is only a hardware token. The term dongle is used for example in combination with Wi-Fi as well. But this hardware can be used to protect software, for example by reading special hardware-specific information.

The role of the dongle will be crucial during the execution of the software: carefully selected parts of the software code will be execute in a trusted environment inside the dongle while the rest will execute normally from the host system.

2.2 Keywords

Intellectual Property protection; copy protection; software protection; software licensing; software piracy; dongle; code extraction; automated code extraction.

2.3 Justification, motivation and benefits

There is an ever-going battle between copy protection and software piracy when it comes to applications as an intellectual property. While the vendors try to protect their products against unauthorized use, pirates do the opposite by circumventing - also called cracking the underlying security mechanisms.

Therefore, it seems impossible to create a security mechanism that would be able to resist forever. A practical solution to this challenge is to create a security system that is not feasible to attack. In the best case, the cost to circumvent the software protection should be more expensive than the price of the software itself and the effort would be as high as recreating the software.

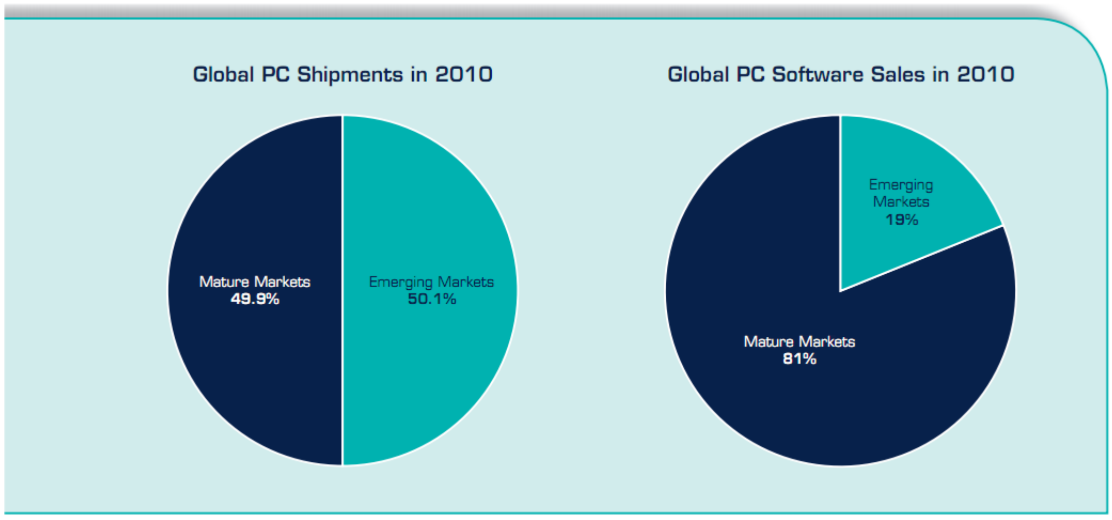

Figure 1: PC Shipments and Software Sales in 2010 [1]

It is important for software vendors to protect their software and ensure that only their authorized users can use it. This is especially true for vendors that produce expensive software packages, where the necessity of protection is bigger, the pirate interest in cracking is higher and the companies' possibility to invest into a more costly solution is certainly better.

Software protection dongles, as one of the more secure alternatives, are being used for some years now. Nevertheless, because of the simplicity and security models used, most of them were broken after a time and the protection level they offer did not show to be satisfactory. Code migration is the new technique, which is supposed to provide a higher level of security in this aspect. Therefore, we aim at creating a new design for the dongle protected software, which will use the code extraction techniques and provide a much higher level of binding (linkage) between the dongle and the protected software. Some solutions that currently exist and claim to provide code execution in their dongles seem to be too weak. Their claims for offering Intellectual Property protection with existing hardware seem to be unsupported by their design.

2.4 Research questions

This project is an advancement in the state-of-the-art software copy protection, but because of the broad topic, I will focus my research trying to answer the following questions:

- Is it possible to create a dongle software protection mechanism that will require at least as much resources to circumvent it as it would to re-create the complete software package itself?
- What mechanism for such a security solution could be appropriate so that it would make it impossible for an attacker to reconstruct the code contained in the dongle by analyzing the communication between the dongle and the software?

- How to generate a code scanning framework with automatic self-extracting capabilities that would be suitable to be used in any software package?
- What is the metrics and a suitable model for evaluating the security of this software protection dongle?

2.5 The contribution

Traditionally, the dongle was aimed to serve in a sense similar to an authentication: if the software verifies the presence of the dongle, than the software could be used, otherwise it would stop its execution. The focus of this project is to extend the role of the dongle and use the dongle to execute a part of the program.

The idea is to perform a cross-compilation step after the code development has been finalized, which will allow the dongle to automatically select and extract parts of the algorithm (code) , where it will later be executed. The challenge is to create such a framework that will enable developers to automatically select (carefully chosen) portions of code and execute these parts in a trusted environment on a micro-processor inside the dongle.

A very important part of the work was to create such a framework, which will be able to scan for the software code, automatically extract part of it and load it into the dongle whilst modifying the code references in the software package at the same time.

The added security here consists of making it impossible for the software to be executed without the code stored in the dongle. Consequently, if an attacker tries to analyze the software by using any of the reverse-engineering techniques, he will fail since the code will never be fully available in the host. Whenever the software would execute such code fragments, the dongle will receive a request along with the necessary data, execute the function on the micro-processor and return the output to the software. This allows only black-box attacks on the dongle, as the attacker has no control over the trusted platform. It should be impossible to recreate these algorithms, if they are vital for the software and unknown to the attacker.

My contribution in this project was choosing the methodology and implementing the automated code-extraction capabilities. In this sense, I worked in defining the framework and implementing such a system, with the aim of being an easy-to-use and compatible with as many platforms as possible. Also, creating a threat model for the security of our solution is part of my job in this thesis.

3 Theoretical Background

Among the different software licensing models that exist today, we have to differentiate between [8]:

Open Source / Free Software is a special software licensing model which does not prohibit software copying, the user is able to download and change the source code. Therefore, the Open Source / Free Software does not need to apply any protection, since it is meant to be free.

Freeware is similar to the Free Software, but its source is not open and therefore it is normally not possible to do changes. Same as the previous category, this software is not meant to be copy-protected.

Shareware usually includes such software, which you can try for free, but with limitations: Some software from this category can be used for free for a limited period only (the so-called free-trial) and a license must be purchased to use it beyond this period. The other type of Shareware contains software which can be used for an unlimited period of time, but it does not contain the full functionalities, which can be used only after upgrading to the full version - buying the respective license.

Node Locked License is a software license model which bounds a software to a specific device (a node). The idea is so that the user pays for every unit (node) where the software is to be used: one licensce, one unit. Such examples include Copy-Protected Games which you can (normally) only play with their original CD/DVD, (High-Cost) Dongle-protected software, and so on.

Floating License is an alternative to the Node Locked Licensing, since it allows for a central management of different licenses for more machines. The idea is that you can buy a single license for a software, which can be installed in a (limited) number of computers and can be used at the same time. A central server in the network is normally used to manage this type of licensing.

3.1 Copy Protection

The licensing models described above need to be enforced in practice. While the free software needs no extra protection, the issue is different with non-free software. Because of its nature, law-enforcement measures are usually not enough to prevent unauthorized copying and use of non-free software. Therefore, technical solutions have been developed to prevent software piracy, which can be cathegorized into two different solutions:

- software-based, or
- hardware-based

3.1.1 Software-based copy protection

Software-based solutions are the simplest form of software copy protection. As their name suggests, they use software built-in functions to validate a certain license. They come in different forms, but the most common software-based copy protection solutions use:

Serial numbers, where a user supplies an input (a *serial-key*), which is a random-looking sequence of characters, during the software installation or at the first software execution. The serial key is generated by some mathematical algorithm, which is usually not as random as they should. Furthermore, an algorithm is used to validate the user-supplied input (the serial key). Therefore, breaking the serial-key generating algorithm or the validation function will make this mechanism useless.

Online Activations use software built-in functions to generate an installation-id and a product key via a hash-function [3]. The software uses the hardware attributes of the host system where it gets installed the software (license) is locked to it.

3.1.2 Online versus Offline validation

Some software vendors use a different scenarion to protect their Intellectual Property from abuse. They require that the software communicates over the Internet with the validating server to check for the presence of the license in the host when the program executes, periodically during the program execution when the user goes online, or some even require that the user is always online to continuously check for the license. Such examples can be typically found in some computer games, which require a continuous internet connection while playing.

It should be noted that the above-mentioned (hardware- and software-based) copy-protection methods can also be combined with the Internet-based copy protection to provide a higher degree of protection.

3.1.3 Hardware Tokens (Dongles) for Copy Protection

Software comes in different types and targets different users. In cases when it contains innovative algorithms which are meant to be kept secret, this is a special kind of Intellectual Property, this needs to be protected.

Most of the commercial software products in the market today apply some sort of copy protection. There are many technologies available for this purpose and each of them has their own implementation, security and use characteristics, but generally all of them fall into one of the two main categories: *local* or *remote* validation [9]. The dongle-based software protection schemes fall into the former category.

Hardware-based copy protection solutions come in different forms and implementations. The main characteristics of this protection system is the use of a special piece of hardware, together with the software functionalities, to validate the given installation. Depending on when the hardware authentication is used, we can distunguish between two main types of hardware-based copy protection systems [8]:

Copy protection based on passive dongles. This type of protection checks with the operating system if the required hardware device is connected to the computer during the installation or when the program starts.

Copy protection based on active dongles. Unlike the passive mode, this model actively checks for the presence of the hardware (the dongle) to prevent software abuse. Our project focuses on this type of copy protection and it will be discussed in more detail in the following chapter.

It is common for the hardware tokens to use the Universal Serial Bus (USB) port, but there are also other solutions that can be implemented through the Line Printing Terminal (LPT), Express Card, SD Card, PC Parallel port, Ethernet port and so on. Therefore, the term "dongle" can be used to mean the device that uses any of the ports to connect to the computer [3].

3.2 Security issues about software-based copy protection

The above-mentioned methods for software licencing have shown to possess weak security features, as they were broken sooner or later. As the attackers had access to the full software in the host, protection by serial numbers was circumvented either by analyzing disassembling the target program, disabling the functions that were used to connect to the validation servers or generating valid-looking serial keys which the servers accepted as authentic [3]. Some tools, such as SoftIce for Windows systems, can be used for this purpose. With this (and other similar tools), one can generate the assembler code for the targeted software and other debugging possibilities. After the extraction of the validation algorithm, it can be bypasses or a key generator can be implemented for that purpose [10].

Similar attacks can be performed on most of the software-based copy protection methods described above. Therefore, a lot of efforts are being put on an alternative measure - the use of hardware-based solutions - dongles.

3.3 Dongle-based protection security

Dongles are pieces are hardware that are used for validating a certain copy of a software. The dongle is produced and shipped together with the software package by the software vendor, thus adding to the degree of the control of the publisher over the specifics of the dongle. The security in the developed mechanisms so far has relied on the verification of the dongle presence during software execution. The software (which is installed in the computer) checks if the dongle is present in the system after it loads in the memory in order to continue its execution [9, 11]. This is the simplest type of the dongles, but it may be circumveneted using different breaking mechanisms.

Attackers have broken such systems by skipping the verification step. They have observed the call to the dongle and the respective response using an always-true answer from an emulated dongle [11, 12] are the most implemented techniques used to break such schemes. The main weakness here is the simplicity of the operations performed in the dongle.

Other, more complex solutions to dongle-based protection systems include the possibility to perform some operations inside the dongle. The software send a pair of input parameters to the dongle and compares the returned result to the expected one [9]. Analyzing the calls to the dongle and dongle's response to the software, attackers have been able to break such systems. Emulating dongles in software and making the software communicate with the emulated dongle, which is capable of performing the same operations as the dongle, has been a successful attack on such systems. Techniques used in this sense include reverse-engineering methods such as code debugging, obfuscation and similar are typical examples of such attacks [12]. Anti-Debugging [13] and anti-obfuscating techniques have been developed by software vendors, but it is only a matter of time until they are reverse-engineered as well [9].

Other important sources of attacks which will be useful for me during my thesis include online reverse-engineering forums with the newest attacks and counter-attacks, such as *Collaborative RCE Knowledge Library* (http://www.woodmann.com) or *The seekers' Windrose* (http://www.searchlores.org).

3.4 Cryptography

Current copy-protection dongles, besided the challenge-response protocol implemented, also employ cryptographic functionalities to provide another layer of security. Encryption is the process used to transform information (the plaintext) into a form which makes it undreadable, except for the person(s) who posses special knowledge to decrypt it. Normally, encryption is performed using a certain encryption algorithm and a key, while decryption is the reverse process of generating plaintext from the cipher-text (text in encrypted mode) in order to make it readable again [14].

3.4.1 Public-key vs. Private-Key Cryptography

Traditional cryptography used to work on the principle of a secret key, which the sender and the receiver of an ecrypted message know and use [15]. The sender encrypts the message and the receiver is then able to decrypt it using the same key. This method is known as *private key cryptography*. This system works as long as the sender and the receiver are the only ones who have knowledge about the key, but the challenge for this system is agreeing on the same key to use for both parties, especially in cases when the two are far away and use electronic communication means to exchange keys. During this exchange, an adversary can intercept the exchanged keys and consequently, is able to read, modify and forge messages [16]. Therefore, managing keys in this system is a challenge (weakness).

To overcome this challenge, *Public-Key Cryptography* was proposed as an alternative. Introduced by Diffie and Hellman [17] in 1976, this method was found to be useful for two primary mechanisms: privacy protection (*encryption*), but also for authentication *(digital signatures)*. The concept is based on the idea that each party in the system gets a pair of keys: a *private key*, which is kept private from a user, and a *public key*, which is published and may be known to the other parties. The need for both parties to share the secret key is elliminated, as all the communication is performed on an message encrypted with a public key, while decryption can only take place if the receiver knows the secret key.

3.4.2 One-way cryptographic functions

One-way cryptographic functions[1] are a very useful tool in cryptography. A one-way hash function is defined as a function F, such that it satisfies the following criterions [18]:

[1]Also known as Manipulation Detection Codes, Fingerprints, Crypto Secure Checksums or One-Way Functions [18].

1. *F* can be applied to any argument of any size. *F* applied to more than one argument, *F* is equivalent to *F* applied to the bit-wise concetation of its arguments.
2. *F* produces a fixed size output (measured in the number of bits).
3. Given function *F* and an argument *x*, it is easy to compute *F(x)*.
4. Given F and a "suitably chosen" (random) *x*, it is computationally hard to find an

   $$x' \neq x$$

   such that

   $$F(x) = F(x').$$

So, hash functions on a given input of any size produce an output of a fixed size (length, i.e. 56 bits). It is easy to compute the hash value of a given input, but knowing the reverse process must be computationally infeasible: knowing the hash value of an argument, it is difficult to find the original input. Randomization functions are used to encrypt the input value in such a way that small changes in input produce big ("unpredictable") changes in the output. Therefore, these functions can be used for Integrity Checks.

4 Related work

While the dongle-based solutions have been studied and used for this purpose before, a limited amount of work has been made in the past in the exact scope as this project. A number of dongle-based copy protection solutions are offered today in the market, but studies and experience has shown that most of them fail to achieve their goals - they posses design or implementation weaknesses which can be exploited.

An early work on this area is a PhD thesis from Kent [19], where he describes the different security models and requirements for, what he calls, "externally supplied software", including the type of software we are interested to protect. He mentions the concept of decentralization and bureaus as agents which serve as an intermediary between the client and the software vendor. In this case, he acknoledges the requirement for a trustworthy and accountable intermediary, which will be used to properly manage and charge the customers. Also, the assumption is on the effectiveness of the measures put in place by the operating system, which enable a decent control over execution of the protected software, but protection from reading or writing over it. Another concept brought by the author in his work is the *mutual suspicion*, where he describes the two different situations: the *hostile host* and the *hostile code*.

Program Evolution is a concept proposed by Cohen in [20]. Aiming the protection of Operating Systems, he studies the basic attack points in software systems and suggests *evolution* as a combination of different defensive techniques. Acknowledging the static nature of the traditional defenses employed, he suggests a dynamic approach. Keeping in mind Shannon's theory of secrecy[21], the author acknowledges the complexity level evolution for performing cryptanalysis on a given target system. Therefore, his suggestion also aims at increasing the difficulty level for breaking the protection mechanisms, taking into account the dynamic nature of attacks.

Providing a selective survey on the software protection approaches, van Oorschot in [22] reviews the literature and brings a number of defensive approaches from software tamper resistance concepts, code obfuscation techniques, software diversity to white-box cryptography.

Gosler in [23] also provides a survey of software copy protection techniques, including dongles. The author brings up different methods for protecting against software analysis, such as anti-debugging techniques, use of checksums and code encryption. In any case, he acknowledges the need to balance the techniques for software analysis and modification resistance. Additionally, Gosler suggests renewing (updating) the software at (regular) periods before the interval required by an adversary to break it, a technique known as *software cycles*.

Further, Herzberg and Pinter [24] propose CPU built-in cryptographic capabilities as another method of protecting unauthorized software copying. Of course, this requirement is more feasible today than it was in the 80s, when they made this proposal.

A paper by Kingpin [25] presents a set of successful attacks on USB hardware tokens, which allow access to user's private data, without having the legitimate credentials. Considering the academic papers published in the field, the author presents mechanical, electrial and software attacks on USB tokens and gives some recommendations about how to increase the security in such areas.

A more thorough study of the attacks on microcontrollers and smartcards is brought by Skorobogatov's technical report in [26]. His PhD thesis brings the list of non-invasive attacks, such as power analysis and glitching, and invasive attacks, such as reverse engineering and microprobing. Additionaly, as the title of the report suggests, he focuses on a new class of attacks, which he calls "semi-invasive" attacks. These attacks stand in between the previous two: like invasive attacks, these attacks require chip depackaging, but they do not require electrical contact to internal chip lines, thus leaving the passivation layer intact. The author claims that this class of attacks represents a bigger threat to hardware security than the other two, as they can be as effective as invasive attacks, but cost as low as non-invasive ones. In the end, he also presents some defence technologies which can (should) be used to protect from the identified attacks.

On the other hand, (Jozwiak *et al.*, 2007) present in [27, 28] two studies where they bring an analysis of the efficiency of the software protection devices with memory and time meters. They also show that attacks on such systems are feasible and an attack is presented for each type(HASP envelope for the former, and ATMega 128 MCU on the latter). They bring up the importance of binding the software and the hardware key. One of their central contributions in this area is their claim that the strength of the pretection offered by hardware keys is directly linked to the dependency level between the protected software and the key: a protected software should absolutely and completely depend upon the presence of the dongle.

Additionally, the Chair for Embedded Security at Ruhr-University Bochum together with Escrypt has done some internal studies for our project. I found it useful to read the Bachelor thesis of Pöpper [29], where the author analyzes the security of some of the current dongle solutions for software protection. Similarly, there were two seminar papers in similar regard. The first one was from (Heggeman *et al.*) in [3] and dealt with protection of software from illegal copying and discussing the security of software with regard to preventing multiple instances of software running at the same time, binding them to a specific target, such as dongles, CD/DVD media and other hardware tokens. The othe seminar paper from Bornhöfft [8] presents state-of-the-art hardware methods for software licensing, including dongle protected solutions and takes a closer look at those with code outsourcing (migration).

5 State of the Art

A dongle is a small device which is externally connected to a host (computer) in order to provide protection against illegal software use. The concept of using dongles for software protection is not new and it has gained more attention during the last three decades, but as with any other solution, it was just a matter of time until they got broken and their protection useless.

Initially, dongles were simple and "dumb", in the sense that the software they protected was merely checking the dongle presence [1] to validate the license. Needless to say, they were easily emulated and therefore, the software could be used without a license. Similarly were the dongles with specific serial numbers broken. Installing a virtual device with the characteristics of the original dongle performed perfectly as a crack.

More intelligent dongles were used later, which possess more functionalities and could also store the license in them, which could then be used for one specific application or for more applications from the same vendor. Their functionalities also differed depending on whether they provided host-based or network-based protection, offering the possibility to use a single dongle connected to a central network server, which can authenticate licenses for several hosts on the network. Cryptographic features that were implemented on these dongles provide a higher level of security.

5.1 Cryptographic features

Most of the dongles in use today are capable of implementing cryptographic functions. One could argue if these dongles can be considered as another group of dongles with security features (so-called Crypto-Dongles [3]) or as an improvement to the group of intelligent dongles with security capabilities. In any case, the security level here highly depends on the cryptographic capabilities, the hardware design and the innovation level of the proposed copy-protection solution.

The simplest form of the dongles with cryptographic features works based on a challenge-response protocol. This can be implemented either as a symmetric or assymetric encryption protocol, but because of the faster execution, most of the dongles use block (symmetric) ciphers, such as AES. To protect the software, some dongles encrypt the communication data that are transferred between the host and the dongle (through an API) or by encrypting important data which are stored in the dongle, such as the license of the product.

Because of the system design, the symmetric key systems in this case are easier to attack since the key is stored both in the host system (where the software resides) and in the dongle. A successful attacker can gain access to either of those keys and the security measures of the dongle will be broken.

[1]By checking if it is connected to the computer

5.2 Code Outsourcing

Another feature which is now implemented in some dongles is the ability to store and execute (parts of) programs in the dongle. This way, a part of the software is outsourced to the dongle, which executes the selected part of the software. In this case, the full code is not included in the host, making it impossible for an attacker to fully reverse engineer it. The outsourced algorithms that are to be executed on the dongle are encrypted for the dongle, which is the only party capable of decrypting and executing it. In theory, the dongle is more trusted than the host and the more algorithms are stored in the dongle, the better security level it provides. In practice, it is difficult to implement a large number of algorithms in the dongle. This is due to the efforts to produce a cheaper price, which needs to be implemented on a simpler hardware. Therefore, there is a trade-off between security and execution speed.

This type of dongles are known to provide a higher level of protection as long as the dongle and the material stored in it is kept secure. Therefore, the key material used for cryptographic operations needs to be kept secure, the firmware authentic and the algorithms secret. If the communication between the host and the dongle is encrypted, this will make the design even more secure.

5.3 State-of-the-Art Dongle Solutions for Software Copy Protection

A number of dongle-based software copy protection solutions available in the market today and each competes with each other claiming higher level of security, performance and adoptability. The ones that are more important and interesting were:

- SecuTech's Unikey Pro,
- SafeNet's Sentinel HASP HL,
- KEYLOK Fortress,
- WIBU Codemeter,
- Feitian's ROCKEY 6 SMART PLUS, and
- Senselock's EL Series.

5.3.1 Unikey Pro

Unikey Pro is a solution developed by SecuTech Solution Inc., a cadanian-based company [30, 3, 31] and it provides a series of UniKey dongles. The UniKey Pro is the one offering the highest protection level, offering network functionalities and is also equipped with real-time clock, password-protected and/or encrypted flash drive.

5.3.2 SafeNet Sentinel HASP HL

Previously known as Aladdin solutions, the Sentinel HASP dongle is now offered by SafeNet, who acquired the former company in 2009. [3, 32, 30]. It claims to offer an automatic file wrapper through HASP Envelope, which provides with encryption, code obfuscation and system-level anti-debugging technology. It supports a number of programming languages (.NET, Java and C for MSVC 8) and can run in Windows, Mac and Linux [32].

Hardware	Unknown IC and an EEPROM chip, Atmel 24cl28w
User Memory	6 KB
Cryptography	AES (128 bit), RSA-DSA
Price	Around 50 EUR per piece

Table 1: SafeNet's Sentinel HASP HL [3]

5.3.3 WIBU Codemeter

WIBU-SYSTEMS offers a number of software and hardware based solutions for applications or documents protection [3, 4]. For software protection, the best solution offered is the Codemeter, which employs a number of protection mechanisms and encryption schemes, as presented in table 5.3.3.

Hardware	Samsung Smartchip - S3 Series
User Memory	384 KB
Cryptographic scheme(s)	AES (128 bit), TDES, SHA-256, RSA (1024 bit), ECC (224 bit)
Price	56,90 EUR per dongle (volume of 100 pieces)

Table 2: WIBU Codemeter - brief overview [3, 4, 5]

5.3.4 KEYLOK Fortress

KEYLOK, an american-based company, offers three software copy protection solutions based on dongles: KEYLOK II, Fortress and S-LOK. The one that is of interest for us is the Fortress solution, which is a Windows USB only that is capable of migrating functions and executing them in the dongle [33]. It provides an extended memory of 5,120 bytes, expandable to 55,000 bytes, which can be used for licensing options, date-based licensing and counters. Also, it provides an

Hardware	Unknown Smartcard chip
User Memory	50 KB
Cryptography	Proprietary Encryption Algorithm
Price	28,25 dollars per piece (for a volume of 100-249 pieces)

Table 3: KEYLOK Fortress

Anti-Debugging Utility (*PPMON.EXE*), which is supposed to prevent debugging of the software. It provides and API for communication between the host and vendors claim to have a secure memory, but offer no further details about the type of security measures implemented. It comes with a manual and a set of tools which are meant to make the implementation easier. It uses proprietary encryption algorithm, which is not made public. Similarly, hardware implementationd details are also kept secret.

5.3.5 Feitian's ROCKEY Series

Dedicated to smart card and chip-based security technologies, Feitian has designed a series of hardware solutions for software protection named ROCKEY. One of special interest for us is ROCKEY6 Smart, which offers a higher level of security and code migration capabilities. It is a 32-bit smart card based dongle and it claims to be cross-platform. The dongle comes equipped with a smart-card, which runs on a Card Operating System with Proprietary IP - FEITIAN COS [6]. ROCKEY7.NET is the next series from Feitian, but it is limited to supporting .NET applications.

Hardware	Unknown implementation
User Memory	70 KB
Cryptography	RSA, DES

Table 4: Feitian's ROCKEY6 Smart [6]

5.3.6 SenseLock's EL Series

Senselock also offers a dongle protection for software with code migration and the vendor also provides a patent about it [8], describing the detailed processes and operating workflow. The patent describes both the workflow of the dongle operating with the protected code imported and the other one mentions some improvements to enhance the performance. Here, the model takes into consideration the limited execution speed in the dongle, they propose a model where the software continues to execute until the current thread has finished (the dongle has returned the computed results), which therefore requires a CPU-equipped dongle.

Senselock's EL Stf contains an NXP (Philips) 16 bit chip, which is certified with an EAL 5+[2] and contains a user memory of 8, 16 or 32 KB.

[2]Evaluation Assurance Level is split into levels from 1-7.

6 Methodology and HIKOS

The project I got engaged in had already started last year (2010) and it is ongoing for one more year. Therefore, there was a lot of useful information I could use and previous studies the project team had made. This made it easier for me to have a starting ground on one hand, but I did not want to limit myself only into this, so I also made paid a certain amount of time to gather information about the available dongle copy-protection solutions on my own. Then, it was easier to understand which direction my task should be focused.

My work for this project I can split in two parts. In the first one, I spent most of the time analyzing the project requirements and getting to know the tools to be used; the second step was defining and implementing a code pre-processing framework; and finally, presenting a security model for the project I was involved in.

The framework I creates is supposed to be platform independent and therefore we decided to implement it in ANSI C using only standard platform-independent commands and functions. This was a smart decision, but since I was not very familiar with C so much, it took some weeks of preparation to start the real implementation. Later, as requirements got more and more detailed, I realized that in some points ANSI C did not define any available functions, so I had to go for platform-dependent solutions. This made some parts of the framework be possible to only execute in certain operating systems. Also, since the framework was supposed to offer maximum flexibility, it should not be bound to a certain language, but support any programming language instead, this made an extra task for defining the right metrics for the syntax analysis. Testing the framework in different platforms was also part of this major step, which took most of the time for the whole thesis timeline.

The last step, security analysis was carried out during the lifetime of the project and a part of it in the end. The team had already done some work on analyzing the security of some available solutions, so I could use some of their previous work. Studying our solution was an extra work I had to carry out and especially since I did not take part in the hardware implementation, I needed continuous feedback from the working team, which was quite helpful. Defining the threat model and attack trees for our solution concludes the work on this thesis.

6.1 Resources and Parties

The project was implemented by two major partners:

- *escrypt*, an international private security solutions provider located in Bochum and their primary focus was in developing the hardware prototype with the agreed security features, and

- *emsec - the Chair for Embedded Security* at Ruhr-University Bochum, where I was engaged and where my working place was located.

The team members (including me) had access to the common SVN repository, which housed a lot of useful information about the project, the previous studies (seminar works, bachelor and master thesis' on specific parts of the project) and other relevant material, which was very helpful to get into the topic more quickly.

On the other hand, the Chair offered a personal computer and the while computers and other IT facilities I used at emsec, the Chair for Embedded Security. I had my personal computer set up and ready to use with both Ubuntu and Windows installed and I used Eclipse CDT for the development environment.

Also, I used the library resources with printed books and online materials from both the Gjøvik University College and the Ruhr University in Bochum.

The project required decent knowledge in what we called "Basic Knowledge" and some Special Knowledge was to be acquired during the project runtime. For the Basic knowledge part, the project required

- knowledge and use of ANSI C without the use of any Operating System specific functions or libraries;
- knowledge in Parser Programming / Regular Expressions; and
- Cross-Compilation and Code Execution on Embedded Devices
- Threat Modelling and Attack Trees
- Dongles and hardware implementations
- Cryptography and secrecy theories

Part of the basic knowledge required for the project is the knowledge about Intellectual Property Protection, such as

- Basic Problems with Software Protection, such as reverse engineering approaches, as mentioned above.
- Hardware Dongles as IP Protection and the difference between Protection through hardware and software.

Special Knowledge requirements for this project include:

- Semi-Automated Code Extraction
    - Problems with automated code extraction, such as ensuring all connected nodes/data is extracted, analyzing data dependencies, preventing side-effects, and so on.
    - Problems with automated identification of "interesting" code parts for extraction (efficiency, security gain, memory consumption, code/data size)

6.2 Products

As a result of my work, I have defined a framework for code extraction, analysis and validation, as well as performed a threat model for the HIKOS. The framework definitions and the implementation is protected by a Non-Disclosure Agreement, as a limitation from our partner - emsec, but the main functionalities and workflow are presented later in the report. The threat model has been developed and is presented in the chapter of security analysis.

The result of the development for the first part is a tool that can be used in any platform, but I have tested in only in Windows, Linux and Mac, with the requirement that the machine offersa gcc support. The tool can be used to scan, extract and analyze a given source code.

6.3 Development Tools

I used the GCC toolchain for my project and I was working under Ubuntu using Eclipse CDT for C/C++ support. In Windows, I tested it in Windows 7 with *mingw* and Eclipse CDT. The program code was implemented in ANSI C, while for presenting the threat model, I used a tool called *AttackTree+*, which is a proprietary software, but I was able to get a license for a short period.

7 HIKOS - The Architecture

Our solution falls into the category of Code-Outsourcing solutions with external hardware - dongles. The main intention is to offer a solution which will offer smart protection of the Intellectual Property which will be highly secure. The innovation consists on the ability to implement in any type of software project and run in any platform; use highly-secure design and highly-secure hardware to provide a safe execution environment; and provide a smart code-extraction and analysis framework to help in the process of code outsourcing.

HIKOS[1] (High Security, Intelligent Software Copy-Protection) is a project (to be fully) implemented by the two partners, *emsec* and *escrypt*, in a period of two years. The main aim of our solution is to offer protection for single instances of programs with innovative algorithms rather than large scale licence management systems. This comes because of our primary interest to protect the intellectual property - the algorithms and the implementation details that are meant to be kept secret. Therefore, our solution to achieving this was by extracting (carefully selected) parts of the program and execute them in a higly secure environment - in our dongle. To help developers for a simpler and quicker extraction procedure, we provide a framework which will enable a semi-automatic code extraction [34].

Following is a description of the general architecture, the general concept and a workflow of HIKOS.

7.1 Security Concept

The main protection target of our solution are single instances of programs with innovative algorithms. Thus, the focus is on protection the intellectual property - algorithms and implementation details - by making the application depend on the dongle. Keeping these algorithms secret is made possible by storing and decrypting them in the dongle, where the secret keys are securely stored, which is the basic idea of our solution. If the algorithms we are protecting are made public, then it is possible for an attacker to reconstruct it and thus this solution will not work. Therefore, the algorithms that we store in the dongle must be secret. Our solution offers a model where an attacker will not be able to have the full software available as long as the extracted parts are crucial and non-trivial.

To have a secure execution, we offer a dongle with high security, which protects both the (parts of the) software and data with high security requirements, such as encryption/decryption keys, hash values and so on.

To protect the software, several (critical) algorithms from the source code are extracted.

The extracted functions must be (ANSI-) C code, because we will cross-compile each extracted function for the dongle architecture. Calls to those functions are replaced by API calls to dynamic libraries. We also provide a solution which handles the communication between the software and the dongle, send/receive parameters and process returned values.

[1]from German: *Hochsicherer intelligenter Kopierschutz für Software.*

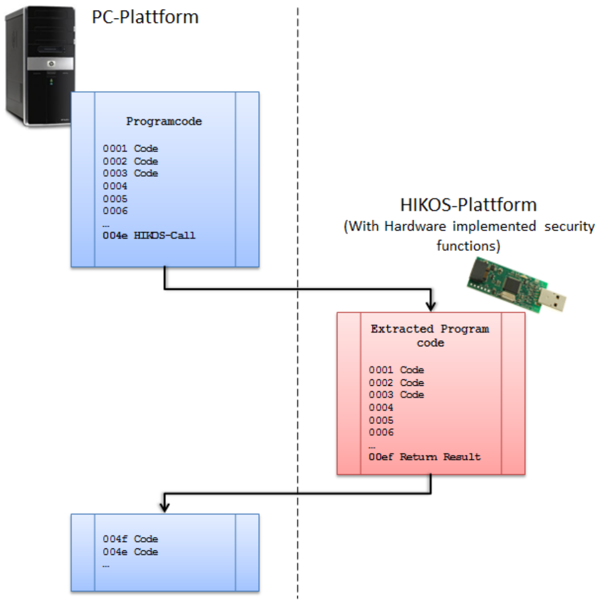

Figure 2: The general working principal for our software copy protection solution - HIKOS

Also, to check for a valid license, a licence generator is used, which encrypts and signs the binary for the specific device.

Following, we will describe our solution based on the description made by the project team in [2, 35, 34]. Because these documents are not meant to be public, I will skip the details.

7.2 Design Workflow

HIKOS suite comes with three main tools: *a Preprocessor, the Runtime Library* and the *License Generator*. The following description is compatible with the current version of the HIKOS documentation [2, 35, 34].

The *Preprocessor* is part of the workflow and it can be integrated into most of the IDEs automatically by using pre- and post-processing project settings. Other than that, it can also be executed as an external tool and it scans the target source code for certain parts of the code. When the "interesting" blocks of the code are found, they are automatically transferred to the dongle (the extraction phase) and the same blocks of code are replaced by calls to the (respective) HIKOS library and they are compiled in the dongle as shared libraries. More on the pre-processor will follow in the next chapter.

The other tool, *the Runtime Library*, is only responsible for handling the communication between the software and the dongle. It checks if the dongle is present before any communication takes place between the two. This does not add to the security of HIKOS, but it is used for exception handling. When the dongle is connected to the computer and the program execution reaches a statement which calls a function from the dongle, a check is performed to see if the dongle contains that function.

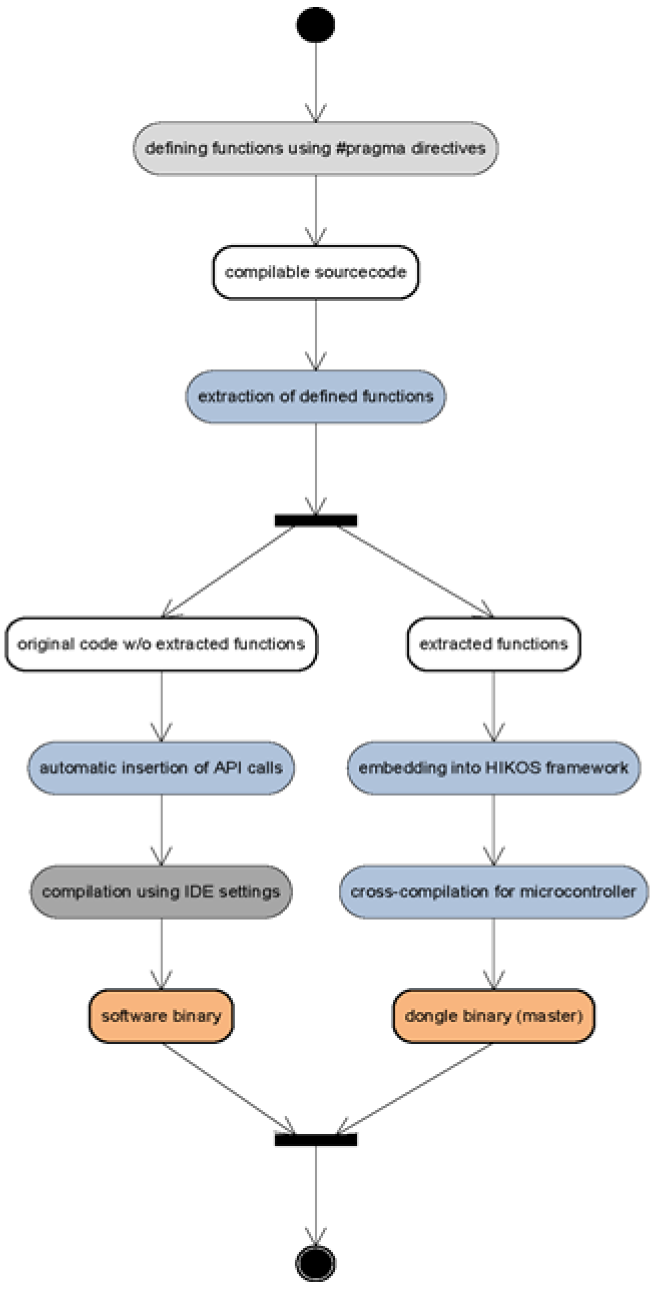
defining functions using #pragma directives
compilable sourcecode
extraction of defined functions
original code w/o extracted functions
extracted functions
automatic insertion of API calls
embedding into HIKOS framework
compilation using IDE settings
cross-compilation for microcontroller
software binary
dongle binary (master)

Figure 3: HIKOS Workflow. The color in each action represent the responsible (person or program): light grey is the developer, blue color is for the pre-processor and the dark grey for the IDE [2]

The extracted functions are handled separately: they are first embedded in the framework and then analyzed by the pre-processor. In cases of errors or uncomplete extracted code, the pre-processor gives out warnings and error messages with the details. After a succesful extraction, a summary of the extraction process is shown to the user.

When the program finishes, it calls a *free* function, which releases the memory allocated for the application on the dongle, as well as the license reserved for the program. The same function can be also called automatically after a certain period of time when the program is not used. However, the exact period has yet to be decided.

7.3 Hardware Architecture

Our solution runs on the assumption that the remote execution environment is kept secure. Therefore, a combination of high-security hardware components has been studied and as many security features have been implemented in hardware, I will briefly describe them.

A simplified model describing the concept of communication between a host system and the dongle is provided in figure 4

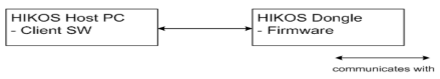

Figure 4: A simplified model of HIKOS

Among the security requirements for the dongle were a minimum of 128 MB flash memory for the application and the possibility to be used as a random number generator. Therefore, we used an ARM 11 processor and an internal security monitor, including a secure internal RAM memory. The microprocessor interacts with the externally connected memory in the dongle via a bus system, which is also used to communicate with the smart card, as shown in figure 5.

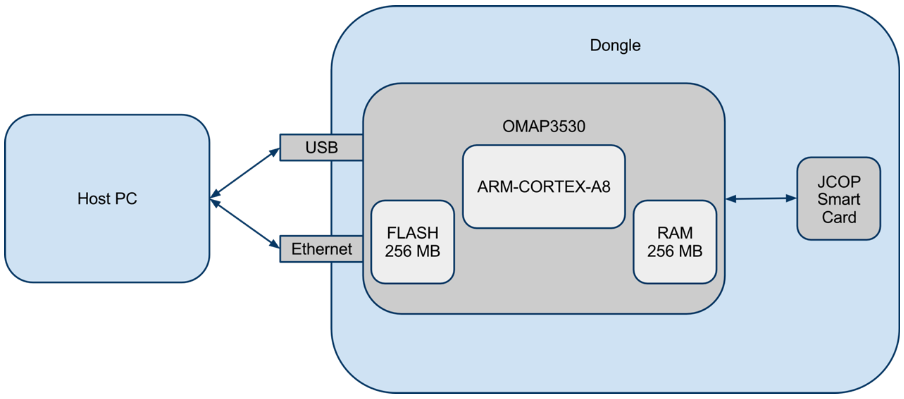

Figure 5: HIKOS hardware concept: a Crypto CPU with Internal Security Controller

This was the initial concept, but because of the unavailability of some components, the current prototype is implemented using a Cortex A8 microprocessor and the smart card is a J-COP card,

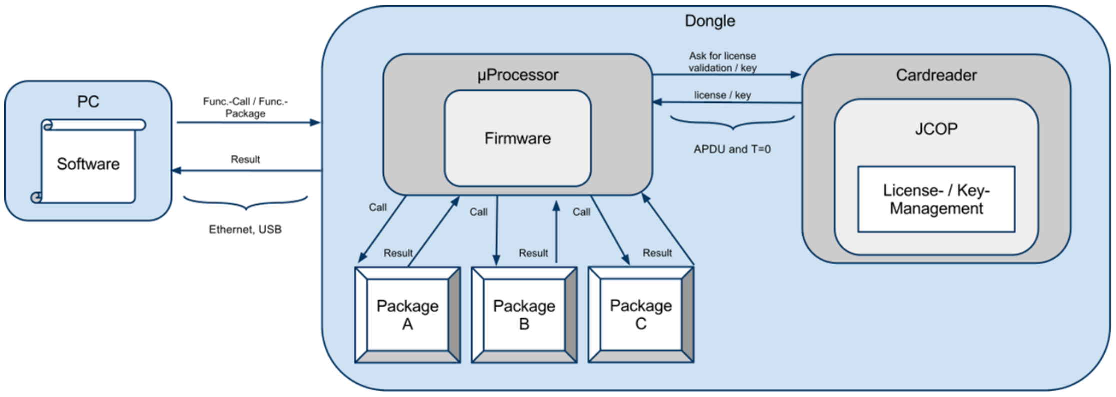

Figure 6: Software Architecture - HIKOS

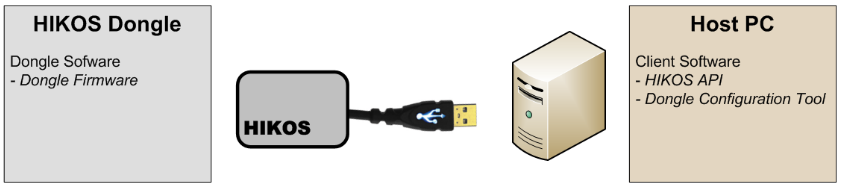

Figure 7: A simple representation of our model - the main definitions

which is certified with EAL 5+.

7.4 Software Architecture

7.4.1 Definitions

In this thesis, the following definitions will be used: *Host PC* represents the end user system, while the protected software is called *client software*. The client software is executed in the HIKOS dongle and interacts with it via the hardware interface implemented with USB and Ethernet connection. The *Dongle Firmware* represents the operating system with special programs installed in the Dongle.

7.4.2 Firmware

The firmware in HIKOS contains a Linux operating system and several software modules, which are used to receive and process information with the software on the host PC, as well as some maintainance tools.

7.4.3 Software Protection Interface

A HIKOS daemon is mapped to different hardware interfaces in order to communicate with the protected application on the host. When the daemon receives a function call request, the message will specify program call request together with the program ID. The daemon then checks if the requested program code is residing on the memory and is available for execution. If the program is not in the memory, the daemon send a message to the host, requesting transmission of the (packages of) code and dismissing the function call request. Otherwise, if the requested function is already available on the dongle memory, the service will validate the license. If this executes with a positive result, the daemon requests data transmission from the host. It then requests from the smart card to decrypt the functions and uses the memory to execute the program with the received parameters from the host. In the last step, either the function returns the processed value or the daemon returns an error code. This concludes a step of a normal communication between the host and the dongle.

To protect from buffer overflows, I have implemented a run-time check which is supposed to protect from such errors. This will be described in the next chapter when I describe the code analysis and extraction process.

7.4.4 HIKOS API

An Application Programming Interface has been designed to enable two very important processes in our model: the *configuration* of the dongle in terms of granting or revoking licenses, and the *interaction* between the dongle and the client software. This API is distributed as a Dynamic Link Library (DLL) on Windows platforms or as a Shared Object (SO) on Linux platforms.

7.4.5 Setup/Configuration

The Dongle Configuration Tool is used to set up the dongle using HIKOS API calls. The first step for such a configuration is the search for the dongle. After "finding" the dongle, the configuration tool checks if it is a valid dongle, by checking the current dongle against the enumerated dongles. This is done through the Vendor ID on the dongle. If the dongle does not contain a Vendor ID, the tool imports the Vendor ID and the corresponding Vendor Key in the dongle. When the dongle has the Vendor ID present, the License Keys are checked. Depending on what the desired task is, the vendor can import new keys or revoke existing keys on the dongle.

In the case of a new license purchase, the tool will export a unique Dongle ID, which is used to bind a specific license to a specific dongle.

7.5 Security Architecture

The license scheme uses symmetric key cryptography, and due to legacy reasons, since older JCOP cards did not support SHA-256, it uses SHA-1 hash for integrity verification. AES is used to encrypt the IP packages exchanged between the host and the dongle, while RSA is used to import the license verification keys and vendor keys, such as AES keys. The initial process starts by the vendor encrypting the extracted code and the customer gets them encrypted on the host.

The license keys are stored in a highly secure storage - the JCOP card, which has a capacity of 80 KB, and they never leave the smart card. For the moment, AES product keys are not stored in the JCOP card, but they are kept in the Cortex A8 (microprocessor) as we are trying to offer a higher performance level (the smart card will probably not provide the desired throughput).

The dongle comes equiped with a private RSA key and a signature of the public key (certificate), which is signed by the dongle vendor (escrypt). The customer buying the dongle can then export the certificate and encrypt his product keys with it. Therefore, the dongle is the only party able to extract the keys.

The current flash does not provide memory with security features, so the product keys are kept obfuscated in flash.

The communication between the processor and the smart card is encrypted, but the hardware packaging is done on a package-on-package model. So, the microprocessor, the RAM memory and the Flash are stacked over each other, protected by another special closed casing. This is meant to prevent an attacker from opening the case and observing the communication from the bus connecting those three.

7.5.1 Secure Boot

The final dongle will contain a security chip[2], which is a fast ARM CPU, but contains internal key storage and a security controller to perform hardware Triple DES. This controller contains eFuses, which are one time programmable and will store the Triple DES key.

Without a secure boot process, an attacker could easily change the firmware in the dongle and then bypass any security checks in the dongle. The other security measures in the dongle would not need to be executed and therefore the security measures in our dongle would be useless.

To protect from such an attack, we use a process we call *Secure Boot*, during which the (highly-secure) chip will validate the firmware. In our model we sign the firmware by a 2048-bit RSA key, which is kept anywhere in the dongle, while the hash of the public key is safely stored in e-Fuses. The internal security control of the security chip will validate the authenticity of the public key (by checking the hash value in the e-Fuses, which is a SHA256 function) and validate the whole firmware. The firmware is then able to use the Triple DES engine to decrypt the AES key for communication with the smart-card and use it for the license validation request.

The security of e-Fuses consists on their characteristics of not being re-writable. Once set, they cannot be altered. In the case of 3TDES key, it can also not be read, except just used by the internal hardware to encrypt or decrypt messages.

License Management for the HIKOS project has not been completely finalized, but a general concept has been create and it has been presented in figure 8.

[2]Due to the NDA I signed with escrypt, I cannot specify the exact model of the chip.

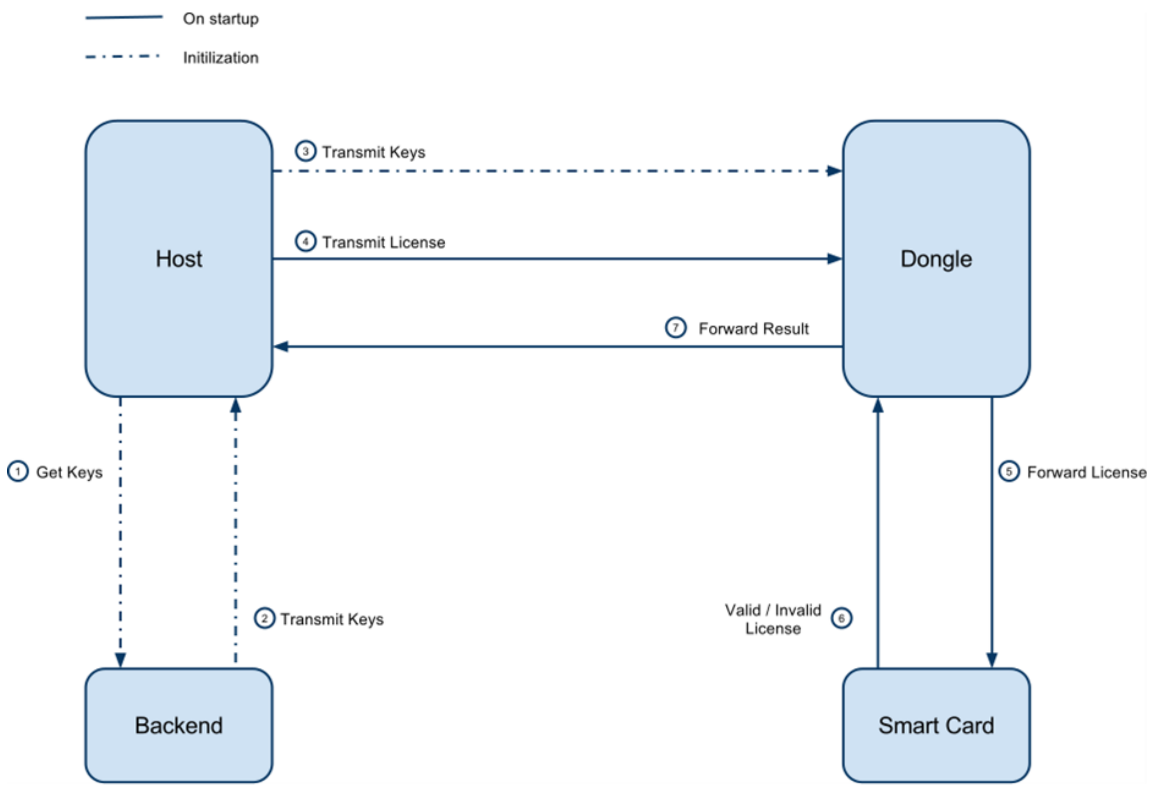

Figure 8: HIKOS License Management Overview

8 Code Pre-processing, Analysis and Extraction

As shown in figure 3, some important steps include Code Analysis and Extraction. I defined the framework and developed it as an independent tool to be integrated with the HIKOS framework.

This task has been implemented as a semi-automated process, during which (a part of) the code is transferred from the source to the dongle. The processo starts by the developer marking the parts of the code that should be extracted and our application takes care of the rest. As a process, it is performed in the following consecutive steps:

Scanning - during this step, the Code Extractor scans the given path recursively searching for the source files. Every line of the source files is scanned for the special keywords, which define the positions where the extraction should start or end. Since the source can be written in any language, the Extractor needs to know what symbol(s) are used by the language to define block symbols. This is important to avoid compilation errors after the code has been transferred to the dongle in the cases when the software is written in other language than C.

Code Extraction is the step when the actual extraction process takes place. Code that is meant to be extracted is moved from the source path to the destination into a single file until the extraction process finishes.

Post-Extract Analysis and Reporting - Assuming that the two previous steps have successfully completed, the framework performs an additional step to check for code validity and unresolved dependencies, as well as "wrap" the imported functions and package them into libraries.

8.1 Source scanning

The first step towards extracting the code from a given source is scanning. When the CodeExtractor is initialized, it receives a source path, where the project code is located. Here, we have to differentiate between two types of scans performed:

(Recursive) Directory Scan is the process of scanning the given directory and all its subdirectories for source files. This is the first level of scanning.

(Source) File Scan consists of steps to read and do the necessary computations for the with the source files. Once a regular source file is found, the File Scanning begins. During this step, the file is read line by line and the application looks for specific (pre-defined) keywords in it. Such keywords include, for example, the keywords for language definitions, language specific symbols, commands for starting/stopping extraction, starting/stopping deletion and so on. The actual commands as used by the framework are shown in table 5.

Keyword	Description
`@hikos_settings`	Signals that the original language definitions will follow in this line.
`@language`	Defines the source language, in which the original code has been written.
`@block_comment`	Defines the block comment symbols, as used by the specified source language.
`@method`	The method definitions are expected to follow after this keyword.
`@param`	Following this keyword, the method parameters
`@extract_start`	Extraction of the code should start after this line
`@extract_end`	Code extraction should stop after this line.
`@delete_start`	Remove the code after this line.
`@delete_end`	Stop removing the code after this line.

Table 5: Some of the keywords used for code-scanning and extraction, as defined for the HIKOS framework

It is important to point out the reasons for choosing such a step-by-step methodology rather than perform everything at a single step. Of course, it could be possible to start extracting from the source code as soon as the scanner matches the `extract_start` command, but due to possible errors or missing configurations in the source file, this could result inefficient. Therefore, I found it smarter to scan the whole file first, "memorize interesting" code segments and validate the source file configuration to make sure everything is safe to run.

As mentioned earlier, it is the developer's responsibility to properly mark the source code with the special keywords, by which the application initializes the necessary values. As the source code can be written in any language, we need to make sure that we know how the language defines comment symbols, functions and parameters, and so on. Also, we scan the whole file to make sure that the number of `extract_start` and `extract_end` commands is following a consecutive flow and matches each other respectively. The same is true for the `delete_start` and `delete_end` commands. If we find inconsistencies, then something might have gone wrong or the developer might have forgotten to properly mark the source code. In that case, the extraction is not executed, but the developer is notified about the errors instead. This is all part of a validation function, which makes sure that all the necessary values have been initialized during this phase and that we are ready to continue with the next step - Code Extraction.

This procedure is recursive in the sense that every directory and subdirectory at the given path will be scanned and the source code analysis will be performed for every source file found.

8.2 Code Extraction

The process of extracting the code from the source is performed upon a successful execution of the previous steps, namely the scanning and validation of the source file. This is the key process and perhaps one of the main aims of the whole tool, but not necessarily the most complicated one. The previous step has already memorized the lines where we are supposed to start extraction from, so in this step we read the given file once again line by line and once we reach the marked positions for extracting code, we copy those lines into an output file (in the previously supplied directory path) until we reach the command for stopping the extraction. This is performed for any `extract_start, extract_end` that were encountered. Similarly, for the parts of code which are marked for deletion, we merely output the complete file without those lines into another output file, close and delete the current file and rename the output file to the original source file name.

8.3 Post-Extraction Analysis and Reporting

After the whole process of extraction has been successfully performed and assuming that the extracted code is stored in a single output file at the given path, we perform a post-extract analysis. During this step, we use different techniques and tools to analyze the different characteristics of the extracted code, apply some protection measures and split the extracted code into libraries.

The analysis we make here is based on the complete set of extracted functions and for every function individually. One of the issues we have to take care is making sure that we offer a good performance for the extracted functions. Keeping in mind that these funcitons will be executed on the dongle, we have to provide some sort of metrics for measuring the complexity of the extracted code. Therefore, we apply our knowledge and use existing tools to calculate this. As a first step, we generate the assembly code for every extracted function. This way, we have an approximation of how many instructions of assembly code will be required to perform the operations in them. We used GCC compiler options to generate and save the compiler output and then created a parser which will scan the output. This way, we could approximate the number of assembly instructions each function will generate.

We all know about the advantages of using loops in a programming language. But, sometimes we have to take care of the maximum and average number of steps a certain loop is executed at every function call. This is especially important when we have to deal with nested loops. Therefore, we also check the extracted functions for such computations. Again, GCC was shown to be very helpful in this case. It performs code optimizations during every compilation. We can save that information into a separate file, which we can generate using GCC. Again, another tool for parsing the contents of this file had to be written. This way, we are able to approximate the number of iterations in the loops and calculate loop depths (nested loops).

8.3.1 Code Packaging

Because the extracted code can occupy large memory blocks and considering the memory limitations in the dongle, we agreed that we must protect the memory from overloading. Therefore, our solution was to separate the extracted functions into different libraries and each of the libraries can be loaded separately in the memory and stored there as long as those functions are required or until the dongle memory get exhausted and we need to replace some library with another one. There were different options available for this, but trying to figure out the one that would perform more efficiently and require less load/unload operations to/from memory required careful analysis. We call this a packaging problem.

The first thing we thought of was defining an upper bound for the size of a single library. Measured in the number of bytes, we could assign a fixed value, up to which we could add functions. The same procedure could be applied to the rest of the libraries, until we reach the end of the code. This is similar to the Knapsack problem, where one looks for the most efficient solution to pack the heaviest things first unti the knapsack is full.

Other techniques were also studied, but we finally came to the conclusion that we need to define smarter metrics for assigning a function to a certain library. To avoid loading and unloading the libraries from the memory too many times, we figured out that it would be smarter to pack inter-dependent libraries together. This way, functions calling other functions could be grouped together into a single library. To achieve this goal, we tried different alternatives to implement this, but finally we realized that we can GCC does some pre-processing and dependency check during the compilation of the source files. Therefore, we were able to save the compiler output and analyze it in a seperate file. This way, we could parse the file contents and use the necessary information.

8.4 Wrapping Extracted Functions - Overflow Protection

During the scanning phase, we collect a lot of information which will be used in the later stages. For example, one of such cases is the function specifications. We use this information to provide some measures for protecting from runtime exceptions or buffer overflows from an unfiltered input. Since every function will be called by some parameters, we thought of developing a measure to check the supplied parameters. For this purpose, we require that the developer defines in advance and clearly marks in the source code every parameter type and length. This is especially important when dealing with strings (`char *` in C). During the runtime, we check if the length of the supplied parameters matched the length of the defined parameters in the source file. In cases of inconsistencies, the execution will stop and the error flags will be raised accordingly to be returned to the host.

9 Security

9.1 Evaluation Criteria

From a security point of view, a suitable concept is the one of *protected subsystems*, introduced as a security requirement by Kent in [19]. It serves as a model for protecting both host-based and distributed software, but our focus is on the former. This model is equivalent to our concept, requiring a higher level of difficulty for an attacker to extract special part(s) of the software *(the subsystem)*. A subsystem is defined by Schroeder [36] as a "collection of programs and databases that is encapsulated so that other executing programs can invoke only certain component programs within the protected subsystem, but are prevented from reading or writing component programs or databases, and are prevented frm disrupting the intended operation of the component programs". Such a system is intended to detect modifications and disruptions by physical attacks, rather than preventing them.

Kent [19] also defines some additional criteria, such as:

Decentralization This criterion requires that the protection mechanisms must be decentralized in order to gain from the advantages it offers from its centralized counterpart.

Effectiveness To meet the security requirements over a broad spectrum of attacks, the protection mechanisms must provide a unified approach. Based on an anticipated threat environment, the protection mechanism should change only its parameters, rather than the system itself.

Generality/Flexibility A decent protection mechanism should be compatible with a wide range of applications, systems and equipment, rather than requiring to rely on a certain technology or equipment, which the system should be depenendent on.

Low Cost A cost-benefit analysis should result positive for the use of the protection mechanism: the cost of using the protection should be lower than the cost of losses caused by not using it.

Good Performance It is important for a solution to be able to perform at an acceptable time, which does not degrade the use of the software itself. A trade-off between effectiveness and performance may be usually required, giving each the necessary position.

Transparency The requirement is that the protection mechanism should be unobtrusive, so that integration with the writers of external software is made with ease and without effecting (much) the design of the external software.

Our project (HIKOS) was developed aiming to offer a secure solution for Intellectual Property protection. But, as with any other security solution, it can only claim security aiming to fulfil certain security assumptions. Even though companies like to promote what they call "secure" solutions, we will suffice aiming a solution which will be regarded as providing a high level of security.

Rather than performing specific attacks on the dongle, I chose the methodology of providiing a formal way of describing HIKOS security, based on a variety of attacks. In this sense, I used the *Attack Tree* model, as presented in the next section.

9.2 Threat Model

By threat model, we try to describe the security aspects of a given system, in our case - the HIKOS dongle, by defining a set of possible attacks on it [37]. A threat model helps better understand the overall security of a system, the probability of certain attacks being performed and apply the necessary countermasures to protect from it.

When it comes to security standards and metrics for software protection solutions, more specifically, dongle solutions for this purpose, there is no proper framework for assuring an acceptable security level or a proper metric for security evaluation [12]. Attampts to offer models for evaluating the efficiency of dongle solutions for software protection have been the attention of some studies and one such model is presented by (Piazzalunga *et al.*) in [12]. Their model aims at presenting a monetized security strenght measurement for software protection dongles, as well as a forecast of the time needed by a hypothetical attacker to break them. Three main macrocomponents have been used as a starting point for this model:

- the dongle,
- the (dongle-protected) software, and
- the host, where this software is running.

Piazzalunga *et al.* [12] propose a model for evaluating the strength of dongle protection. The methodology their propose consists of building a *defense patern catalog* and an *attack pattern catalog*, where we list the applied protective measures and the possible attacks, respectivelly. The results of the model are presented in an attack-tree model, with the node values specifying the time required for an attacker to perform the attack specified, respectivelly.

I found the model adequate to start with and considering the specifics of our solution, I create an adapted model based on [12].

9.2.1 Defense Pattern Catalog

A defense pattern includes the main attributes about the strength of the protection mechanisms implemented [12]. Adhering to the specified principles by Viega and McGraw [38] for effective piracy protection: scattering license management software, obfuscation of code, integrity verification and misuse detection, we can also add the principle of *dongle authentication*, as to protect from communicating with an emulation software. The following defense pattern catalog adheres to the above-mentioned principles, which in our case can be grouped into:

AES Challenge-Response. The data exchanged between the host and the dongle are encrypted with AES. A random-number generator is built in the dongle hardware, so challenge-response works with a large keyspace, making it more difficult for an attacker to find the key. This relates to challenge distribution, since the large keyspace and the randomization

Defense Pattern	**Attribute**
AES Challenge-Response	Challenge distribution Number of AES keys used Randomization Large Key space (to avoid repeatibility)
Memory Usage and Protection	Number of different caller addresses Memory address separation On-fly decryption
Communication channel protection between the CPU and the Smartcard	Hardware 3DES encryption Highly-Secure DES key storage (EAL 5+)
Linkage to dongle libraries	Static link to libraries High dependency level
Integrity verification	Firmware integrity verification Read-only firmware
Highly Secure components	One time programmable e-fuses Read-only portions of memory Execute only permissions Misuse detection and self-destruction

Table 6: Defense pattern catalog for HIKOS

function used prevent key repeatibility.

Memory usage is a protection used to detect the dongle authenticity. If we assume that an attacker has been able to emulate the dongle, then this measure that we send different challenge into different locations in the memory to make sure the dongle responds respectively. Also, it provides protection against fake libraries. The effectiveness of memory usage is measured through the number of different caller addresses.

Communication protection. There are several protection measures in place to prevent communication leakage between the CPU and the smart card. Only if the Internal Security Control in the CPU is able to provide the required information to the smartcard is then possible to retrieve the encrypted AES key.

Linkage to dongle library. The dongle can easier be emulated if there is no strong connection between the client software and the dongle. Since we use code migration, the software is strongly dependent on the protected code, which only the dongle can decrypt.

Misuse detection Also, the hardware components provide capabilities for detecting when debugging tools are being used, which in turn calls a delete function to erase the content of the memory and the smartcard.

Integrity verification Integrity of the firmware in the dongle is checked by the secure boot process, as described in the previous chapter, and ensures that the verification steps are performed in the dongle, as long as it is intact.

Attack pattern	**Steps**
Tamper with memory	1. Locate a library call in the software 2. Use debugging tools to capture password 3. Analyze and modify memory content
Emulate dongle A)	1.Recover the protected libraries 2. Develop the code for emulation library 3. Replace calls to the dongle libraries to the emulated dongle libraries
Recover the protected libraries A)	1. Recover the AES key 2. Decrypt the libraries
Recover the protected libraries B)	1. Use debugging tools to read the dongle memory during execution 2. Retrieve the content of the memory
Recover AES key	1. Observe the communication between CPU and smartcard 2. Get the encrypted AES key from the smartcard 3. Obtain the TDES key and decrypt the AES key

Table 7: Attack patterns and respective steps

9.2.2 Attack Pattern Catalog

On the assumption that both the client software and the host it is residing on may be hostile (untrusted host), an attacker could perform the following attacks:

Reverse-engineering attacks. In this category, an attacker would try to decrypt the encrypted parts of the software.

Dongle Emulation. Once the previous attack has been successful, the attacker can easily emulate the dongle. This emulation would be used to communicate with the client software instead of the original dongle, therefore replacing the dongle itself.

Fake library attacks. In cases when recovering the full software is impossible, an attacker can emulate the dongle and make the software communicate with the fake libraries (not the dongle).

Memory Tampering. Memory on the host can be tampered with and sensitive data, such as passwords, can be retrived from an attacker through the use of a debugger. As a result, the attacker can gain access and communicate with the dongle and already has access to the client software.

The attack pattern catalog and the steps required for each pattern is presented in table 7.

9.2.3 Attack Tree

Introduced by Schneier in [39], *attack trees* has proven to be a convenient tool of describing security of systems and to systematically categorize the variety of ways these systems can be attacked [40].

The visual representation of threats and attacks to computer systems in a tree structure, consisting of diagrams with multiple levels [41] has made this tool helpful for security practitioners and analysts [40]. These trees consist of a single (common) root on the top, which represents the attack goal, while the different ways the attack can be achieved are represented as leaf nodes.

In a given attack tree, we can find AND nodes and OR nodes. While AND nodes represent different steps towards achieving the same goal, OR nodes represent alternatives which can be used to achieve the attack goal. Furthermore, each of the nodes is assigned a value, which can be used to calculate the security of the goal. One type of values is calculated by determining possible and impossible nodes. To calculate the final value of the nodes, we perform the respective AND or OR operation between all the children of a specific node. The value of an OR node is possible if *any* of its children is possible and it is impossible if all of its children are impossible. On the other hand, the value of an AND node is possible if all the children of the node are possible, while it is impossible if any of its children is impossible, meaning that any of the identified attacks is impossible to perform [39].

The model presented in figure 9 shows the different ways the dongle can be attacked. AND connections must all be implemented for an attack to succee, while the OR branches are alternatives to implement a certain attack. Each node can be assigned a value, be it the time necessary for the attack to succee or the probability for the attack to be performed considering the defense mechanisms in place or the design of the dongle.

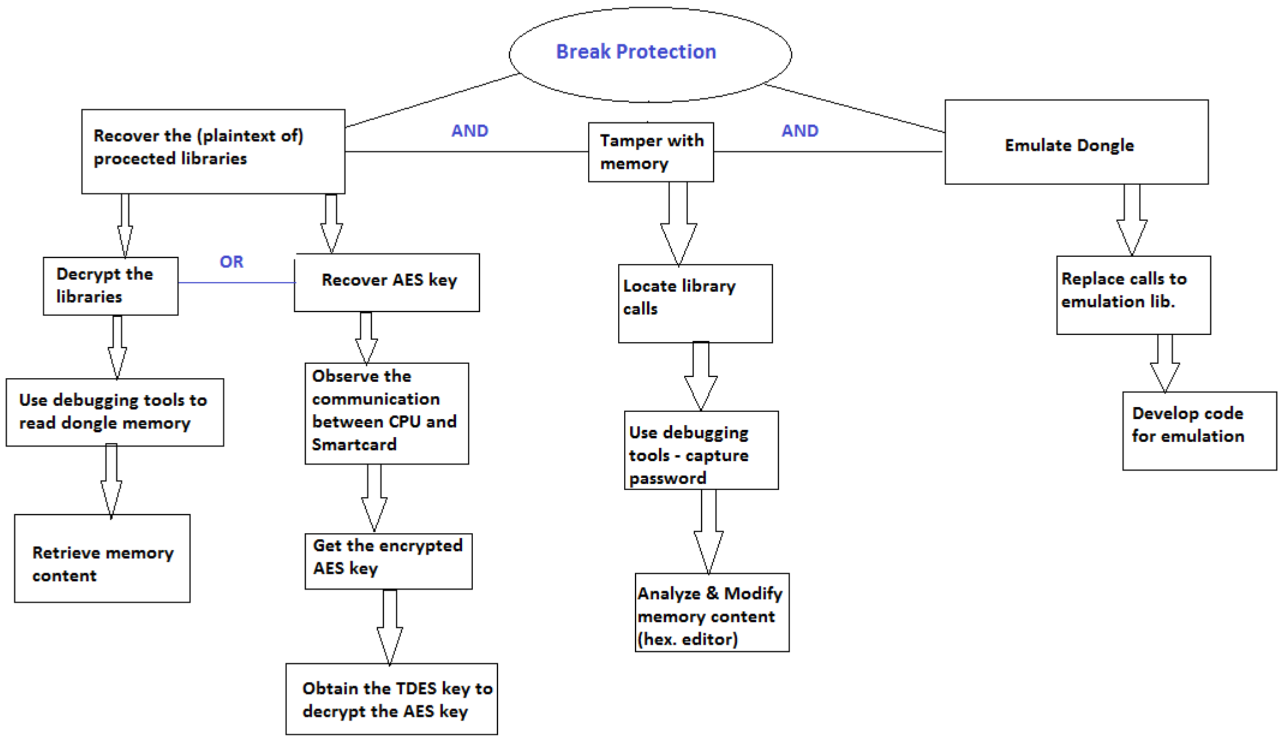
Break Protection
Recover the (plaintext of) procected libraries
AND
Tamper with memory
AND
Emulate Dongle
Decrypt the libraries
OR
Recover AES key
Locate library calls
Replace calls to emulation lib.
Use debugging tools to read dongle memory
Observe the communication between CPU and Smartcard
Use debugging tools - capture password
Develop code for emulation
Retrieve memory content
Get the encrypted AES key
Analyze & Modify memory content (hex. editor)
Obtain the TDES key to decrypt the AES key

Figure 9: Attack Tree Model. The Arrows represent consecutive steps to be performed

For each given node, we can assign the values we need in order to provide a better view of the overall security, the weakest links, and perhaps add preventive countermeasures where necessary. A useful metrics in our case could be the probability that any of the attacks presented here may happen, the amount of time (efforts) required for an attacker to break the security measures, or the monetary value necessary to implement an attack. Building such values requires experimental setup with different software using the dongle, time and tools to perform attacks, which in the case of this thesis, could not be performed. However, this can be used as a good basis for the vendor (escrypt) to measure the overall security of their solution and future work could use it and finish it. A validation of field data could then be performed to check the precision of the model.

10 Conclusion

The work presented on this thesis contains a short, structured description of the problems in the field of Intellectual Property Protection, respectively Software Copy-Protection. The focus of the project has been developing a better, more secure way of protection software with the use of dongles. The concept is not new, but we add to the current solutions, in regards to higher security level both in the design of the (architecture of our) solution in principle and a practical more secure implementation of a prototype.

Security is a paramount to such a protection system and therefore this project gives special attention to implementing state-of-the-art methods and technology in fulfilling security requirements as good as possible, with the aim of providing a solution that can work in any system and development environment. Building on the best practices and filling some of the gaps in the security of previous weaknesses, we believe that the work presented here can be used as a good starting point for further research and analysis. Our implementation of a dongle with highly-secure architecture, components and design is a contribution in this field and we hope that it will be useful in protecting the Intellectual Property of many companies, thus providing a higher level of revenues and investments in software innovation.

11 Future Work

Considering the amount of time during which this Master Thesis was performed, the work presented here brings the results of carefully studied state of the art in the field of dongle protected software and provided a security analysis for HIKOS, including a threat model.

As far as the work on the code-preprocessing is concerned, it provides a framework which can be further tested to different platforms, but its functionalities can also be extended. The current solution I worked on works only with C functions. This can be considered as a limitation for certain software solutions, where object-oriented solutions may be used to implement more complex algorithms to be protected. Therefore, including support for C++, as a language that supports objects, can require certain modifications and can be a good point for future work.

On the other hand, the threat model presented here is a basic foundation, which can be built upon and needs to be validated with field data and the right metrics, which again, needs to take into account the time, effort and monetary value of the attacks that can be performed, together with respective probabilities. Consequently, a set of respective countermeasures can be suggested wherever the results of the validation show necessary.

Bibliography

[1] Alliance, B. S. May 2011. Seventh annual bsa/idc global software - 2010 piracy study.

[2] Zimmermann, R. Hikos specification - software. Technical report, Ruhr University Bochum, Chair for Embedded Security, December 2010.

[3] Heggemann, C. & Markhoff, S. Hardware method for software product identification - dongles, cd-dvd media manipulation and other techniques. Seminar paper, Ruhr-University Bochum - Chair for Embedded Security, January 2010.

[4] WIBU-Systems. Codemeter as token. http://www.wibu.de/codemeter.php?lang=en.

[5] WIBU-Systems. Codemeter asic. http://www.wibu.de/codemeter.php?lang=en.

[6] Feitian software protection dongle. Technical report, FEITIAN, 2010.

[7] Manoharan, S. & Wu, J. 2007. Software licensing: A classification and case study. *International Conference on the Digital Society*, 0, 33.

[8] Bornhöfft, M. January 2011. State-of-the-art hardware methods for software licensing. *Seminar paper*, Chair for Embedded Security, Ruhr University Bochum.

[9] Genov, E. 2008. Designing robust copy protection for software products. In *Proceedings of the 9th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, CompSysTech '08, 49:IIIB.14–49:1, New York, NY, USA. ACM.

[10] Wikipedia. 2009. Softice. http://en.wikipedia.org/wiki/SoftICE.

[11] Spesivtsev, A., Krutjakov, A., Seregin, V., Sidorov, V., & Wegner, V. October 1992. Software copy protection systems: structure, analysis, attacks. In *Security Technology, 1992. Crime Countermeasures, Proceedings. Institute of Electrical and Electronics Engineers 1992 International Carnahan Conference on*, 179 –182.

[12] Piazzalunga, U., Salvaneschi, P., Balducci, F., Jacomuzzi, P., & Moroncelli, C. November 2007. Security strength measurement for dongle-protected software. *IEEE Security and Privacy*, 5, 32–40.

[13] Gagnon, M., Taylor, S., & Ghosh, A. 2007. Software protection through anti-debugging. *Security Privacy, IEEE*, 5(3), 82 –84.

[14] Wikipedia. June 2011. Encryption. http://en.wikipedia.org/wiki/Encryption.

[15] RSA-Laboratories. What is public-key cryptography? http://www.rsa.com/rsalabs/node.asp?id=2165.

[16] MS, A. Public key cryprography.

[17] Diffie, W. & Hellman, M. E. 1976. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6), 644–654.

[18] Merkle, R. C. 1989. One way hash functions and des. In *Proceedings on Advances in cryptology*, CRYPTO '89, 428–446, New York, NY, USA. Springer-Verlag New York, Inc.

[19] Kent, S. T. *Protecting Externally Supplied Software in Small Computers*. Phd thesis, Massachusetts Insitute of Technology - Laboratory for Computer Science, 1980.

[20] Cohen, F. B. 1993. Operating system protection through program evolution. volume 12 of *Computers & Security*. Elsevier Science Publishers.

[21] Shannon, C. Communications theory of secrecy systems:. Technical report, Bell Systems Technical Journal, 1949.

[22] van Oorschot, P. October 2003. Revisiting software protection. In *Information Security, 6th International Conference*, Springer-Verlag, ed, Proceedings, 1–13.

[23] Gosler, J. 1985. Software protection: Myth or reality? In *Advances in Cryptology - CRYPTO '85*, 218, S.-V. L., ed, 140–157.

[24] Herzberg, A. & Pinter, S. November 1987. Public protection of software. volume 5.

[25] Kingpin. October 2000. Attacks on and countermeasures for usb hardware token devices. In *Proceedings of the Fifth Nordic Workship on Secure IT Systems Encouraging Co-operation*, 35–57. Reykjavik University.

[26] Skorobogatov, S. P. Semi-invasive attacks - a new approach to hardware security analysis. Technical report, University of Cambridge, 15 JJ Thomson Avenue Cambridge CB3 0FD United Kingdom, April 2005.

[27] I. J. Jozwiak, A. L. & Karczak, K. 2007. Hardware-based software protection systems: Analysis of security dongles with memory. *IEEE*, International Multi-Conference on Computing in the Global Information Technology.

[28] I. J. Jozwiak, A. L. & Karczak, K. 2007. Hardware-based software protection systems: Analysis of security dongles with time meters. *IEEE*, 2nd International Conference of Dependability of Computer Systems.

[29] Pop̈per, C. Security analysis and evaluation of hardware tokens for ip protection. Bachelor Thesis. Chair for Embedded Security - Ruhr University Bochum, September 2010.

[30] SecuTech-Inc. Unikey software protection, licensing and distribution. http://www.secutech-inc.com/marketing-materials/UniKeychure-En.pdf.

[31] SecuTech. The unikey dongle. White paper, SecuTech Solution Inc.

[32] Sentinel hasp - product brief. Technical report, SafeNet, October 2010.

[33] Software piracy prevention system - keylok ii, keylok fortress. User's manual, MAI Digital Security (KEYLOK), May 2010.

[34] Zimmermann, R. Hikos specification - license management. Technical report, Ruhr University Bochum, Chair for Embedded Security, December 2010.

[35] escrypt. Auswahl und spezifikation der hikos plattform im rahmen des verbundprojektes hikos. Technical report, December 2010.

[36] Schroeder, M. D. & Saltzer, J. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. Phd thesis, Massachusetts Insitute of Technology, September 1972.

[37] Wikipedia. Threat model. May 2011.

[38] Viega, J. & McGraw, G. 2002. Building secure software: how to avoid security problems the right way.

[39] Schneier, B. December 1999. Attack trees - modelling security threats. http://www.schneier.com/paper-attacktrees-ddj-ft.html.

[40] Mauw, S. & Oostdijk, M. 2005. Foundations of attack trees.

[41] Wikipedia. August 2010. Attack tree. http://en.wikipedia.org/wiki/Attack_tree.

A Code-Preprocessing Algorithm

`Input: @source_path, @dest_path`

1. Copy the given source from `source_path` to `dest_path`
2. Recursively scan the directory at `dest_path` and read every file
3. For every (regular) source file, do 4 - 8
4. Perform `Source File Scan` Algorithm
5. If `Validation` B algorithm performs successfully, do `Extraction` D
6. If `Post-ExtractionE` is performed successfully, do 7
7. Perform `Packaging`
8. End.

B Validation Algorithm

1. If Steps 2 - 6 are true, return `true`.
2. The Source language has been defined
3. The Comment Symbols have been defined
4. Methods and parameters have been initialized (defined)
5. Extract start and stop positions match (validate)
6. Delete start and stop positions match (validate)
7. Else do 8
8. Raise proper flags and Report Errors. End. `False`

C Code Scanning Algorithm

1. For every line of code read, do steps 2 - 7
2. Save language definitions positions
3. Save Language symbols definitions
4. Save Method(s) definitions, parameters and sizes
5. Save Positions for Code to be extracted
6. Save Positions for Code to be deleted
7. End.

D Code Extraction Algorithm

Input: @source_file_path, @dest_file_path

1. If Validation Algorithm B returns `true`, steps 2 - 8
2. Read the Code line by line
3. Append the extracted code into `@dest_file_path`
4. Copy the contents of the source code, without the code-to-be-deleted into a (temporary) file and with blank the functions' bodies
5. Delete the source file
6. Rename the temporary file to the original source file's name
7. Insert API calls to protected code into functions' bodies
8. End.

E Post-Processing Analysis and Reporting Algorithm

```
Input: Source_path, Dest_path
1. Report the number of extracted lines of code.
2. For the complexity analyxix, do steps 3 - 11
3. Generate Assembly code for the extracted functions
4. Report the number of assembly lines of code generated
5. Optimize code and calculate nested loops complexity
6. Report loops depth and average number of cycles
7. To Perform Correlation (Dependency-check )analysis, do 8 - 10
8. Use code optimization to generate correlation analysis
9. Save the optimization into an output file
10. Parse the content of the optimized output and save caller and callee functions
11. Report a summary of analysis.
12. End
```