Software Security Techniques
[Internal Report, COSIC]
Version 1.2

Jan Cappaert, Brecht Wyseur, and Bart Preneel
KULeuven/ESAT/COSIC

{jan.cappaert,brecht.wyseur,bart.preneel}@esat.kuleuven.ac.be

October 2004
Contents

1 Problems of Software Protection ............................. 3
   1.1 Black-Box and White-Box Security Model .................. 3
   1.2 Malicious Software versus Malicious Hosts ................. 4

2 Attacks on software ........................................... 6
   2.1 Analysis .................................................. 6
   2.2 Tampering ................................................ 6

3 Software Protection ........................................... 7
   3.1 Hardware Based versus Software Based Security .............. 7
   3.2 Evaluation Criteria ....................................... 8

4 Software Security Techniques ................................. 9
   4.1 Client-Server Solutions .................................... 9
   4.2 Code signing ............................................... 11
   4.3 Code Encryption .......................................... 12
   4.4 Code Diversity ............................................ 13
   4.5 Code Obfuscation .......................................... 14
      4.5.1 Code Transformations .................................. 15
      4.5.2 Pros and Cons ......................................... 16
      4.5.3 State of the Art ....................................... 17
   4.6 White-Box Cryptography ................................... 17
      4.6.1 White-Box Transformations ............................ 18
      4.6.2 Pros and Cons ......................................... 19
      4.6.3 State of the Art ....................................... 20
   4.7 Tamper Resistant Software ................................ 20
   4.8 Software Guards .......................................... 21
   4.9 Oblivious Hashing ........................................ 24
   4.10 Comparison and Conclusions ................................ 24

5 Code Transformations ........................................... 26
   5.1 Classification ............................................. 26
   5.2 Where to Apply Code Transformations ....................... 26
      5.2.1 Source to Source Transformations ..................... 28
      5.2.2 Code transformations during compilation .............. 28
      5.2.3 Low Level Transformations ............................ 30
   5.3 Who will apply transformations and why .................... 30
6 Applications

6.1 E-commerce and E-banking Applications ........................................ 34
6.2 Cryptographic Software ................................................................. 34
6.3 Digital Rights Management and Watermarking ............................... 34
6.4 Embedded Systems and Smartcards ................................................. 34
6.5 Software Agents .................................................................................. 35

7 Conclusions ......................................................................................... 36
1 Problems of Software Protection

More and more software gets distributed by the internet. Examples are e-banking applications, browsers, e-mail clients and also Java-applets, macros, . . . Furthermore, more software and platforms become mobile. Notebooks, wireless devices such as PDA’s and cell phones are becoming part of our daily lives. By all means is an adequate security required in this complex, heterogeneous environment.

First, most of software contains secret, confidential or sensitive information, for example medical files or credit card numbers. To protect this data there exist encryption and authentication algorithms [34], but these again introduce secret keys which have to be protected somehow. Secondly, we have the code of applications. It might be necessary to secure a certain software implementation to guarantee save execution of the program. And third, there is the execution of programs where critical code is executed and some confidential data is accessed. During execution one needs to protect the executing code and the accessed data from malicious intends, such as eavesdropping on secure channels and analyzing static and dynamic code. These three elements all need to be sufficiently secured to guarantee data confidentiality and save program execution to the user.

The major problem in the context of software security emerges when software is distributed to remote hosts. Once this is done, the owner virtually looses all control over the product. And from that moment on malicious users or malicious software [2, 32] can abuse the local software. Chow et al. called these type of attacks white-box attacks [40, 41], because in this model the attacker has full privilege access to the system. This means that the malicious user or program can execute the program at will, observe the memory, processor and registers, change bytes during execution, . . . Therefore, a thorough protection against analysis and tampering of code is desirable.

Furthermore, we notice the evolving threat of malicious programs such as viruses and worms that exploit certain flaws in commonly used operating systems or widely used application. Certain software packets are vulnerable for these attacks and others are not, but due to the current ‘monoculture’ in operating systems a single security flaw can cause major problems and have a big impact on the worldwide network [36, 31, 25].

1.1 Black-Box and White-Box Security Model

Today more and more software is spread through the Internet. Additionally, next to local client software we also use more often mobile software, e.g. mobile software agents, Java applets, . . .
Delays are undesirable during distribution, operation and network communication of current software and therefore people use fast and efficient cryptographic techniques. This is the reason why symmetrical key operations are preferred: their speed. To sent keys in a secure way, people often use asymmetric cryptography. More on symmetric and asymmetric cryptography can be found in [34].

As a consequence distributed software contains keys from which secrecy and integrity have to be protected. Examples are browsers with embedded certificates, e-banking applications with secret keys, ... Securing these distributed software applications needs to be done in a clever and save way, which requires extra security measures. Mobile software agents for example have to be sufficiently secured against malicious hosts [17, 44]. And next to the problem of securing local software, a person also needs to trust the server or source origin of the software. This problem is often solved by authentication methods between client and server.

The model in which client software is attacked by local malicious software or by a malicious user is called a white-box environment. In this model the attacker has full access to the system on which the software is running. A simple solution to circumvent this problem is server-side execution. In that case the software itself is not distributed, but only the services. The user then communicates to a server which hosts the service and thus locally runs the software. In this case we speak about a black-box model because the user has no full privilege acces to the server, but only acces to the running service.

1.2 Malicious Software versus Malicious Hosts

Since long ago the computer industry is focussing on how to protect hosts against malicious client programs [32] by for example developing virusscanners and firewalls. Especially the last year this theme got some extra intrest because of a number of successful, new viruses and worms and the emerging spyware [2]. To protect itself against malicious programs, one will typically restrict the number of actions a client program is allowed to execute. For this way of prevention numerous techniques are invented, for example sandboxing [37, 38].

Due to the rise of the Internet and open networks, researchers started in the nineties with investigations of malicious hosts or users. In this case, a trusted client program is attacked by a malicious host. This can be by a user with malicious intend or by other malicious software on the host, for example a trojan horse. An example of this kind of attacks could be a user who is trying to crack a downloaded application. In this context, ‘crack’ can mean extracting secret key information, information on algorithms or code, making
malicious changes in the code of the application, . . . Several techniques such as fault analysis [9] or searching for stored or embedded keys [24, 46] are easily applicable in white-box environments and very effective when client programs are not additionally protected against it.
2 Attacks on software

Two most commonly feared attacks on software are tampering and reverse-engineering. Tampering are attacks that aim to change the functionality of the software while reverse-engineering techniques try to analyse the software. Both attacks are further discussed in the next paragraphs.

Software attacks can be either static or dynamic. In a white box environment all these techniques can be used. For this reason, software security requires eminent revision. The only thing that might hold an attacker using these techniques are time and resource constraints. This means that if it takes a lot of memory and computing power to analyse a certain piece of code, this code has a higher practical security to resist attacks.

2.1 Analysis

Software reverse-engineering mainly consists of analyzing software. Depending on the intent of the attacker, one can extract hidden algorithms, secret keys and other information embedded in the software. In our attack model an attacker has full control of the host system and can therefor perform either static analysis techniques or dynamic analysis techniques. Static analysis is applied on non-executing code and comprises disassembling, decompilation [16], control flow analysis, ... Dynamic analysis is performed on executing code. To this group belong emulation, debugging, memory dumps, ...

2.2 Tampering

Tampering attacks go one lever further compared to analysis attacks. Typically, one first needs information on the internal operation of the software before one can successfully tamper with the software. Therefore, tampering is most often preceded by several reverse-engineering techniques. And once sufficient analysis has been performed, the attacker will try to change the software to its own needs. For example, a hacker can make software or a service available for everyone, even without authentication or rights to use the software. Analog to analysis, tampering can also be done statically or dynamically, this means on static or on executing code.
3 Software Protection

Software can be protected in many ways. It can rely on trusted hardware, which is *hardware based* protection. Or it can rely on its own implementation and the underlying software, which is called *software based* protection. Some techniques combine both. For example, a in a layered security model, the critical applications can rely on a secure operating system, while the secure operation systems runs on top of trusted, cryptographic hardware.

In the first part of this section we discuss the difference between hardware and software based protection mechanisms. Further in this report we will mainly focus on software based protection techniques.

The second part of this section we state some criteria which facilitate analyzing the security needs of an application.

3.1 Hardware Based versus Software Based Security

The main advantage of software based protection techniques is the low cost and compatibility with existing systems. The main disadvantage is that better techniques have to be invented due to evolving methods that circumvent application security. Furthermore, techniques to either secure either attack software are sped up by increasing computing power of processors and growing capacity of storage media.

Local software and sensitive data can be secured by encryption and authentication. Whenever the software executes it will decrypt its internals on the fly. Unfortunately, this method only guarantees full security if encryption and decryption is done in hardware, for example by use of a cryptographic co-processor [53]. If encryption and decryption is not done in hardware, it can intercepted once it is sent in the clear to memory or processor.

A second hardware technique to protect software is tamper resistant packaging. In this case the software and data are physically shielded from attacks. State of the art in these hardware techniques are cryptographic processors and smart cards. Nevertheless, even in this case some attacks are able to extract some information out of these systems, namely by analysis of side channels such as time and power analysis [27, 28, 39]. Borst *et al.* define a number of techniques to secure against attacks on side channels of smart-cards [12].

The two major disadvantages of hardware based security technology is the high cost and the incompatibility with the base of current open computer platforms. By ‘cost’ we mean the expenses for buying and installing on the one hand and the cost for upgrading and maintenance on the other hand.

In general, hardware solutions offer less flexibility dan software solutions.
Therefore, research should be as hardware and platform independent as possible, so that it offers maximal flexibility to the users. On the other hand, we want to deliver the same amount of security as hardware solutions to resist practical attacks nowadays.

3.2 Evaluation Criteria

The level of security in an application consists of the required resistance of the application against reverse engineering and/or tampering attacks. By means of some model parameters we can specify this level in more detail:

- **Vulnerability**: open systems, such as a desktop, a notebook or a mobile device are much more vulnerable to attacks than closed systems, such as servers behind a firewall.

- **Value of content**: depending on the kind of application and her content (code and/or data) varies the type of attacks and the number of resources used for attacking the software.

- **Content lifetime**: content or properties with a longer lifetime require a higher level of trust and security.

- **Security life cycle**: the security of an application can be designed to be periodically renewable. Systems without (safe) upgrade possibilities need a higher security level than systems with systematic upgrades.

- **Sensitivity for global attacks**: global attacks are attacks concerning a whole system. This is feasible when code contains a ‘global secret’, for example a constant key or data at fixed location for each user. In this case the attacker is able to develop an automated attack and spread it through the Internet.

The actual security level is always a trade-off between the need for security and the means to implement this security.
4 Software Security Techniques

In the next sections we try to summarize techniques designed to protect code against malicious users and programs. This can be protection against either analysis or either tampering.

The techniques are more or less presented in chronological order, while some techniques might still be used in practice and others not or rarely.

4.1 Client-Server Solutions

One of the earliest techniques to protect critical software was to keep it running at the owner side instead of the user side. Critical software was not distributed to unreliable hosts, but maintained on a well protected server. The protection of the server depends on as well network, hardware and software security (e.g. the operating system). The code itself is often not protected by any other techniques. By this setup the services are distributed and not the software itself, as shown in figure 1. From an attacker’s point of view the server will be seen as a black box that can be accessed by sending queries and receiving replies.

Figure 1: The client-server model only distributes access to services but not to the code, which is running at the server side.
The main disadvantage of client-server systems is that the server or the network bandwidth becomes a bottleneck, causing services to be temporarily unaccessible. Although, this can be solved by mirroring and upgrading network infrastructure, a new model has been proposed, called partial client-server (see figure 2). In this model, the sensitive code is split into a critical and a non-critical part. The critical part needs to be protected and is therefore run at the server side, the non-critical part is distributed and is run at the client side. The advantage is that the load of the service is now better distributed over the clients and the server. The code running at the server side can also be substantially smaller, although some extra overhead is needed to maintain communication between the client part and the server part. This directly indicates the main problem. At first glance this model seems to unload the server, nevertheless, in practice the client part and the server part have a highly interactive communication so that once more the bandwidth becomes a bottleneck.

Figure 2: The partial client-server model splits code in a critical and non-critical part. The critical part is run at the server side. The non-critical part is run at the client side.

Although client-server was one of the first and still commonly used techniques to protect software from attacks, it actually circumvents the problem
4.2 Code signing

Some languages (for example C) have no security mechanisms in line that check code before execution, therefore, these languages in particular are very susceptible to tampering attacks changing the program in a way that its computations can not be trusted any longer.

To avoid tampering of a program, its code needs to be protected during transmission and storage. Each time the program executes it should check and verify its integrity to detect tampering. Signing techniques [34] are most suitable for this type of checking. The owner can sign the software and the user can verify the signature appended to the software. This model is shown in figure 3. This is already the case with some Windows drivers that are signed by Microsoft and verified by the user at installation time [35]. One could extend and automate this process so that the signature is verified at each execution of the program. For example, software guards [13] do not sign the code with a key, but verify a calculated checksum with a stored one (more information in section Software Guards).

![Diagram of Code Signing Process]

Figure 3: Code signing allows a user/program to verify if the code is coming from the signing identity and to decide whether it is tampered with or not.

The disadvantage is that without extra security measures in place the code and the signature are still vulnerable to manipulation. If the signature
scheme is known, one could simply change the code to its own needs, re-
compute the signature and replace the old signature by the new one. The 
verification module would then just verify the new signature and would not 
suspect any tampering. The main reason for this vulnerability is that the 
signature and the verification module are not signed themselves.

4.3 Code Encryption

Additional to code signing (see section Code Signing) we can also encrypt 
code during transmission and storage (see figure 4). Tools such as crypto-
graphic wrappers encrypt the code of a software application in order to avoid 
attackers gaining access to the software. It protects software against static 
reverse-engineering and tampering attacks. For example, an attacker cannot 
see the code and therefore not make any structured changes when the code 
is stored on a disk or transmitted over a network. Note that an attacker can 
always flip random bits and what will result in a corrupted application.

![Diagram](image)

Figure 4: Encrypting code is done during transmission and storage. But once 
the code has to be executed it needs to be decrypted.

During program execution parts of code will be decrypted ‘on the fly’ 
with a secret key. Unfortunately, at that moment the code appears in the 
clear, in memory for example, so that it is able to intercept. The intercepted 
code can then be debugged, decompiled, ... This is de main vulnerability 
of this technique and furthermore makes the presence of a secret key this 
technique less suitable for distribution.
Even if the code or the data remains encrypted (research on this has been done by Sander and Tschudin [43]) an attacker can observe what happens during runtime if bits in the encrypted code or data or flipped. This technique is also known as fault analysis [9].

Encrypted and polymorphic viruses [51, 52, 57] perform similar techniques. An encrypted virus encrypts at each new generation the body with a unique key. This is mainly to avoid detection through string analysis searching for specific byte signatures. In front of the body a decryption routine is added to ensure that the virus body gets decrypted on the fly during execution. Nevertheless, if the encryption routine remains unchanged, scanning for signatures is still possible. For that, encrypted viruses evolved and added a mutation engine ensuring that for each new generation also the decryption routine has changed. This kind of viruses are therefor called polymorphic viruses. Note that the decryption routine can of course be protected with other analysis tackling techniques. Once a virus is decrypted and stored in memory, it will choose a new key, encrypt the new variant and add a modified decryption routine. Figure 5 shows the main structure of those viruses.

![Figure 5: Encrypted viruses and polymorphic viruses also encrypt their virus body to avoid signature detection. Polymorphic viruses even permutate the decryption routine.](image)

4.4 Code Diversity

The last months viruses and worms [32] become a hot topic in the media. Triggered by these virus outbreaks discussions often mention the choice of
operating system. This actually refers to the problem why viruses spread so successful. One reason could be that the software community is evolving to a ‘monotone’ distribution, meaning that most people use the same type of operation systems, containing the same type of bugs. This is one of the reasons why viruses, who most of the time try to exploit only one bug at a time, are so successful.

Without arguing about safe operating system design and implementation, we can state that just as in nature diversity is stronger to resist threats such as viruses and worms. It also offers an extra protection against global attacks because once software images are diversified, a common attack might be a lot harder to set up and only parts of the software community might be vulnerable.

Forrest et al. sketches the analogy between diversity in computer systems and diversity in biological systems [21]. Guided by this idea computer code could be randomized, without changing the functionality or loosing much user friendliness or performance. Their paper presents some preliminary results on randomizing stack layouts by increasing certain slots with a random times 8 bytes. Such a simple modification could harden a program instance better against standardized buffer overflow attacks.

Another technique to battle buffer overflow attacks, called address obfuscation, is also based on the idea of code diversity [42]. This technique randomizes the code and data sections on the stack by randomizing all the base and start addresses, locations of routines and static data and introducing gaps between objects.

More on buffer overflow protection techniques can be found in [20, 47].

4.5 Code Obfuscation

Object oriented programming is applied everywhere because it offers numerous advantages to read, adapt or extent code. However, this way of programming in modules leaves many traces into an executable and reverse-engineers will exploit these traces as good as possible to reconstruct the original source code [16]. Therefore, programmers developed several techniques to maximally obscure the internals of a program so that analysis becomes very hard. The most general technique to do this is code obfuscation. This technique applies one or more transformations to code that make a code more resistant to analysis and tampering, but preserve its functionality. Obfuscated code can then be distributed to untrusted hosts without risking to be reverse-engineered soon. This model is also drawn in figure 6.

Code obfuscation is applied more and more due to the need for embedded software protection. It is initially designed for languages such as Java because
Figure 6: The code obfuscation model. Obfuscated code is fully run at the user side but has to be well protected against deobfuscation attacks.

Java bytecode is very sensitive for code analysis. This means it is easy to recover original Java source files out of Java bytecode files. Many Java obfuscators [26, 48] (and deobfuscators) have therefor been designed. Also .NET obfuscators [49] are becoming common on the Internet. Nevertheless, C/C++ obfuscators are very rare and hard to find. Although, C and C++ are very common and widely used languages.

Wroblewski [56] and Mambo et al. [33] propose code obfuscation on an instruction level, e.g. Assembly code. This has certain advantages. First, the code does not have to be compiled anymore, which facilitates integrity checks and hashing of code. This is one of the reasons why software guards [13, 22] are implemented on an assembly level. Second, transforming on an instruction level instead of on a high level is often preferred for watermarking [50].

4.5.1 Code Transformations

Commercial obfuscation programs often only scramble identifier names and remove redundant information, such as debug information, in code. This is quite trivial, but obfuscation offers a lot more possibilities. A good obfuscation exists out of one or more program transformations that transform a program’s control and data flow in a way it becomes harder to analyse and reverse engineer. Though, the only restriction for these transformations is preserving the functionality of the original program. Thus, obfuscation is
a collection of many techniques that are useful for program transformation, obfuscation or randomization.

Furthermore, most of these code transformations are not one way and it is hard to decide where to use which transformations. Therefore, several parameters measure the quality of a transformation suitable for code obfuscation:

- The main restriction remains preserving the program functionality.
- The main goal of code transformations is maximal obfuscation of the original program.
- A transformation needs as much resistance to automated attacks.
- A transformation needs to be as stealthy as possible, as well for static as dynamic analysis techniques.
- Increase in code size and execution time need to be minimized.

Nonetheless these techniques do not guarantee waterproof security, a combination of several transformation techniques can lead to sufficient practical protection against reverse-engineering and tampering attacks.

4.5.2 Pros and Cons

The major advantage of software obfuscation is the low cost and the flexibility of this technique. Depending on the need for security an application can be obfuscated accordingly. In other words: the extra cost and computation time, introduced by the transformations, can be specified by the software owner or programmer in advance.

The most important advantages of obfuscation are:

- Protection: against static and dynamic analysis attacks.
- Diversity: possibility to create different instances of one software application, to battle global attacks.
- Low cost: low maintenance cost due to automation of the transformation proces and compatibility with systems.
- Platform independency: obfuscation can be done on high level code so that platform independency is preserved.

The most important disadvantage are:

- Cost: every transformation introduce extra cost in memory and computation time necessary to execute the obfuscate program.
• Security: obfuscation does not provide waterproof security.

The theoretical realization of obfuscation is funded by the NP-hard results of Wang [54] and the PSPACE-hard results of Chow et al. [15]. Nevertheless, Barak et al. [8] points out that a ‘virtuele black-box generator’, that can hide code of every program, cannot exist. Barak’s model is rather focussing on specific groups of functions and therefor this does not mean that practical secure obfuscation is impossible.

Just like encryption in practice never guarantees perfect protection, obfuscation can not guarantee that an algorithm or the source code will never be extracted. The essence of obfuscation is to guarantee that as well people as automated programs are not able to decompile a program within a realistic, specified time. In this way, it is guaranteed that an application won’t be used anymore if a ‘cracked’ version shows up and malicious uses are forced to halt reverse-engineering attempts due to the need for computing power, memory or special hardware.

4.5.3 State of the Art

Currently, numerous research deals with code transformations, watermarks and tamper resistant software [19, ?]. It is crucial that transformations neither change the functionality, neither destroy a watermark.

Obfuscation is possible at several levels:

• high level (source code), e.g. C/C++ of Java,
• intermediate level,
• low level (object or machine code), e.g. Assembly.

Intensive expertise in Java bytecode obfuscation exists [18, 30] and also in obfuscation of object code [56, 45], but a there still is a lack in C/C++ obfuscation. Obfuscation of C/C++ code is limited to small ‘obfuscation contests’ [5], although C and C++ are very popular languages in the world of software development, for example in open source software. Furthermore, C/C++ has no additional built in security mechanisms such as buffer overflow protection. Java is a type specific language with built in security mechanisms [3] So in the context of certain programming languages extra security to design critical software is required.

4.6 White-Box Cryptography

Since a few years, attacks have been presented to extract key information out of RSA and even DES implementations. Boneh, Demillo and Lipton have
published a method for RSA [11], Biham and Shamir have extended this
method for DES [9]. These new attacks focus on the extraction of the secret
key embedded in a cryptographic implementation, and are a new threat in
security.

In August 2002, Chow et al. defined this new thread model, the white-box
attack context or malicious host attack context as following:

- Full-privileged attack software shares a host with cryptographic soft-
  ware, having complete access to the implementation of algorithms;
- Dynamic execution (with instantiated cryptographic keys) can be ob-
  served;
- Internal algorithm details are completely visible and alterable at will.

The attacker’s objective is to extract the cryptographic key, e.g. for use
on a standard implementation of the same algorithm on a different plat-
form. Obfuscation alone does not help against this threat, because obfus-
cated cryptographic algorithms store parts of the secret key in the malicious
hosts memory and can thus be extracted.

Chow et al. proposed a new technique to secure cryptographic algorithms
against white-box attacks, called white-box cryptography. This technique is
founded on the idea that an encryption function $E_k$ with key $K$ can be
replaced by an equivalent function $E'_K = G \circ E_K \circ F^{-1}$ in which $F$ is an input
encoding and $G$ is an output encoding. The strength of this substitution is
that none of the implementation components computes the function $E_K$ for a
key $K$. An attacker would first have to analyse $E'_K$ and isolate the encoding
functions $F$ and $G$ before he can analyse $E_K$ to find the secret key $K$.

Thanks to the introduced functions $F$ and $G$ it is possible to inject suffi-
cient ‘randomness’ in the implementation so that locating and extracting the
key is becoming hard. So far the only practical disadvantages of white-box
cryptography are the code size and the extra execution time.

4.6.1 White-Box Transformations

To make extraction of secret key information inside a cryptographic imple-
mentation difficult, we can apply several techniques. The general idea is to
spread the secret key information over the whole implementation, and so
extending the cryptographic border and forcing an attack to understand a
greater part of the implementation. The current techniques only apply on
cryptographic algorithms constructed with lookup tables, XOR functions and
permutations. This restriction is not a bad limitation as most cryptographic
algorithms like DES and AES are constructed with these functions.
Given a cryptographic algorithm with embedded key, we will transform this algorithm into a series of lookup tables by using partial evaluation, I/O block encoding, combined function encoding, by-pass encoding and split path encoding. Secret key information is now contained in some lookup tables. By insertion of mixing bijections, the secret key information is spread over all the lookup tables.

A mixing bijection can be defined as a bijective affine transformation $M : GF(2)^n \rightarrow GF(2)^n$ which maximizes the dependance of the output regarding the input. This means that a small change of input, even one bit, results into a maximum change in output. The theory of bent functions can help us a lot in constructing such bijections.

Given a part of the series of lookup tables $L_1 \rightarrow L_2 \rightarrow L_3$, we can insert random mixing bijections $M_1, M_2$.

Given random bijections $M_1, M_2$, we can transform a part of the series of lookup tables $L_1 \rightarrow L_2 \rightarrow L_3$ into $L_1 \rightarrow M_1 \rightarrow M_1^{-1} \rightarrow L_2 \rightarrow M_2 \rightarrow M_2^{-1} \rightarrow L_3$. We define $L'_2 = M_2 \circ L_2 \circ M_1^{-1}$ as the encoded version of $L_2$. Applying this to the whole implementation gives us a series of encoded lookup tables, which represents the same function, but is locally secure.

A property $E$ is locally secure if all know information does not leak any secret information about $E$. When a function $P$ is bijective (like a lookup table $L : GF(2)^n \rightarrow GF(2)^n$), then its encoded version $P'$ is locally secure. The means it is impossible to find any useful information by studying $P'$ because given a lookup table for $P'$, any bijection $P$ is a possible candidate. This is the similar idea of the one time pad, which provides perfect security.

The problem now is to provide global security, and find a mathematical proof for the (non-)possibility of white-box cryptography.

4.6.2 Pros and Cons

The advantages of white-box transformations are:

- Embedded key: secret key information is spread over the boundaries of the cryptographic function, forcing an attacker to understand a bigger part of the implementation.
- Diversity: because of the randomness inserted into the implementation, we can create a lot of different instances of one software application.

Unfortunately, current implementations still have some disadvantages, mainly in performance:

- Execution: a significant increase in execution time, reducing the overall performance of the program.
- Code size: current lookup tables imply also a significant increase code size.
- Security: the security of white-box techniques is still unknown and unproven.

Unlike obfuscation, it is unknown if perfect protection can be guaranteed. There is know proof white-box cryptography is possible or not.

4.6.3 State of the Art

White-box cryptography is a very new topic. Research in white-box cryptography started in 2002 with the publication of a white-box DES implementation by Chow et al. [40], soon followed by an AES implementation [41]. Before, it was believed it is impossible to protect a cryptographic implementation against attacks in the white-box attack context because the executing malicious host has all the information concerning the algorithm and made computations.

At this moment it is still not clear if white-box cryptography can provide practical security in a white-box context. Shortly after the publication of the DES implementation, an attack was found by Jacob et al. [23], and recently also the AES implementation has been broken by Billet et al. [10].

Use of white-box cryptography can provide some extra level of security, but there still is no proof. Research is going on to construct a mathematical basis for white-box cryptography and prove its security. Main disadvantage of using this technique is its increase in implementation size and decrease in performance. Recently Link et al. published a paper with some new techniques to improve performance and security [29]. Unfortunately, white-box cryptography is still not useful for the applications it is designed for, e.g. smart cards, due to a lack in performance and size overhead. It can only be applied in environments with enough resources, e.g. for DRM schemes. New techniques are being researched.

4.7 Tamper Resistant Software

Tamper resistant software requires very skilled programmers working on a binary or source code level to embed ‘booby traps’ for tamper detection in software. A good tamper resistant code always has a dual function. First, the code needs to detect undesired changes and second the program needs to fail in case of tampering.
In ’96 Aucsmith wrote one of the first papers on tamper resistant software (TRS) [7]. He proposed a tamper resistant software architecture which bundles many of the previously mentioned techniques in order to realize a tamper resistant software implementation. His technique combines four principles:

1. Disperse secrets in both time and space.
2. Obfuscation of interleaved operations.
3. Installation unique code.
4. Interlocking trust.

These principles have also been used as a bases for ideas such as code diversity, software guards, code obfuscation, . . .

Aucsmith’s architecture consists out of two parts namely *integrity verification kernels* (IVKs) and an *interlocking trust mechanism*. An IVK is a small, armored segment of code to embed in a larger program. The IVK has mainly two functions:

1. Verifying the integrity of code segments of programs
2. Communicating with other IVKs in order to accomplish these functions securely.

The general structure of an IVK is shown in figure 7. It is organized in cells, which are decrypted at runtime and thus determine the smallest level of granularity which is ever exposed unencrypted. The encryption of cells is done in a pseudo-random order based on generator function. Furthermore, each IVK contains one or more keys. A secret key to sign and a public key to verify signatures made on other code segments.

The second part of the TRS architecture is the interlocking trust mechanism. It consists out of IVKs, an integrity verification protocol and a system integrity program. These three components work together in an interlocking trust mechanism based on mutual integrity verification. An example is illustrated in figure 8.

For more details about the IVK and the interlocking trust mechanism we refer to [7].

### 4.8 Software Guards

Chang *et al.* define small pieces code that checksum code fragments [13]. Calculating an integrity checksum can be done by for example CRC [55]. Using a complex, nested network, these *guards* are able to verify each other’s
Figure 7: General structure of an integrity verification kernel (IVK)

First cell (not encrypted)
- Entry point
- Generator
- Decrypt & Jump

Encrypted cell

Signature operation
Other operations
Accumulator
Decrypt & Jump

Figure 8: Verification of 2 programs through the interlocking trust mechanism.
code plus the program code itself and repair it if necessary. In this way, tampering of the program is expanded to detecting the complete agent network, this means identifying, localizing and eliminating the whole network of guards and then tapering the actual program code itself. A guards graph and its placement in a control flow graph (CFG) is shown in figure 9.

![Diagram of guards and their actions](image1)

Figure 9: (a) A graph of guards and their actions (b) A possible mapping of the guard network to a CFG.

The disadvantage of this method of software protection is that it is hard to automate and thus depends mainly on one’s programmering skills. As a consequence the maintenance cost will be very high. Furthermore this technique does not offer any protection against dynamic analysis attacks.

New research from Horne et al. tries to extend and automate this technique [22] to improve tamper resistance of programs. Their techniques is based on testers and correctors. The testers, code in Assembly, are inserted at source code level, while the correctors are inserted in the object code. The values of the correctors and some watermark values are computed at installation time, resulting in a watermarked, self-checking, fully functional program. For more details we refer to the paper of Horne et al. [22].
4.9 Oblivious Hashing

As a reaction on the concept of software guards checking only static code, Chen et al. propose a oblivious hashing (OH), a technique that allows implicit computation of a hash value of the actual execution [14]. The idea is to hash the execution trace of a piece of code, allowing to verify the run-time behaviour of the software. Hashing instructions are interweaved with the original code and take results of previous instructions and apply them to hash values stored in memory (see figure 10). Assignment results and control flow results captures most of the dynamic behavior of a program, for that it is sufficient to hash only assignments and control flows.

Figure 10: Oblivious hashing instructions are interweaved with original code and hash results of previous instructions in memory.

Oblivious hashing has two major application domains. First, it can be used to provide local software tamper resistance and second, it can be also be used for remote code authentication. In a white box model, local software should provide in its own security so that remote code authentication is not an option. For more details on oblivious hashing we refer to [14].

4.10 Comparison and Conclusions

In this paragraph we rate all the mentioned methods according to the protection it supports against analysis and tampering attacks, both static and dynamic. We did not include “client server solutions” and “code diversity” in
the table, because the former circumvents the problem of local software protection, while the latter is not focussing on a single software instance, but merely on a population. Furthermore, techniques that prevent buffer overflows though code or execution diversity, do not focus on analysis or tampering attacks.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Static</th>
<th>Dynamic</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code signing</td>
<td>N</td>
<td>N</td>
<td>F</td>
<td>N</td>
</tr>
<tr>
<td>Code encryption</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>Code obfuscation</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>White-box crypto</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Tamper resistant software</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Software guards</td>
<td>N</td>
<td>N</td>
<td>P</td>
<td>N</td>
</tr>
<tr>
<td>Oblivious hashing</td>
<td>N</td>
<td>N</td>
<td>P</td>
<td>P</td>
</tr>
</tbody>
</table>

Table 1: N = None, P = Partial, F = (almost) Full
5 Code Transformations

5.1 Classification

According to Collberg et al. there are four main classes of transformations [18]:

- Lexical transformation: names of variables can be swapped or replaced by names without any semantical value. It is even possible to use heuristics to recall values in order to mislead reverse-engineers.

- Control flow transformations: introducing special predicates allow making the control flow of a program more complex while preserving the functionality. By multiple nesting and introduction of ‘dummy’ code dynamic analysis of a program becomes more complicated. [15].

- Data flow transformations: several transformations in data structures are possible. For example storage, encoding, aggregation and ordering of data can be easily changed.

- Preventive transformations: all transformations that make specific de-obfuscation techniques more difficult succeed.

5.2 Where to Apply Code Transformations

Although research in code obfuscation is steadily increasing the last decade, the question still remains: how and where should we apply obfuscation? In this section we will describe several models in which code transformations are applied during software development. These could be applied either by the software owner or either by the user, depending on the scenario. In the next section different types of scenarios will be sketched.

Because in this paper we focus on code obfuscation within the context of software security, we will not focus on code optimization. Code optimization also consists of certain code transformations and adaptations but of another type to optimize a program or its memory usage.

We distinguish the following three code transformation models:

Model 1: pre-compilation A programmer wishes to ship a self made program, but wants to add an extra layer of security to the most critical section in the program (e.g. authentication checking code). In the context of C/C++ and other high level languages, it is easy to write a tool that performs certain source to source transformations without changing the program’s functionality. The transformed source code can either be distributed
and compiled at the user side or either be compiled at the owner side and distributed afterwards. In both cases an attacker analyzing the code will first have to defeat the extra added layer of security.

**Model 2: at-compilation**  This model contains transformations that can be applied during the code compilation process. To achieve this, one only needs to write a new compiler that is non-deterministic or that reads extra input determining where and how code should be transformed. In case no extra input is given, the compiler itself should decide which transformations to use. In this way we are able to generate diverse object codes instances starting from one source code instance. The advantage of an adapted compiler is that it can be easily distributed, giving the users the ability to perform transparently obfuscation transformations themselves.

**Model 3: post-compilation**  Analogue to source code obfuscation, we can apply transformation techniques to obfuscate compiled code. Compiling source code results in object code or some intermediate language code (e.g. Java bytecode). Although, the syntax of compiled code is simpler and less
detailed than source code, it still leaks information, e.g. understandable instructions and procedures. For example, consider an application delivered through the Internet or on CD-ROM. Usually those software packages do not include the source code of the application. To add extra security, people could transform the executable code to a higher level code (e.g. Assembly) that is easier to understand and rewrite. Although Java bytecode still contains lots of high level information, Java bytecode obfuscation can be considered as a post-compilation transformation because bytecode is already the result of Java source code compilation.

In our classification model we clearly describe three distinct models, but numerous combinations are possible as transformations can be applied sequentially on a piece of code and on different levels during the code translation process. In the next sections we will describe these three models in detail.

5.2.1 **Source to Source Transformations**

Source to source obfuscations are very common group of transformations. This group of transformations also includes preprocessors. Consider for example the C preprocessor. This is actually a macro processor that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs. This contains four facilities: inclusion of header files, macro expansion, conditional compilation, and line control. The preprocessed code will in this example be again C code, stating that we actually can call it a source to source transformation.

Nowadays many source to source transformations exist [4], but they are mainly used to optimize programs in speed or size [1].

5.2.2 **Code transformations during compilation**

Compilation of code has become more than just translating a computer program to an executable. Programs are often written in a high level language because of two main reasons. First, the design can be done in an object oriented way which facilitates programmers to maintain, adapt and extend their code. Second, a high level language often offers a large set of standard classes containing predefined structures and functions. By using these classes a programmer can save a lot of time and make sure that the called functions are optimally or securely implemented. These and several other reasons might have the side effect that resulting programs become quite large and slow.
This is why ‘code compilation’ implicitly includes lots of optimization techniques varying from removing death code, optimal register allocation and efficient mappings to the architecture’s instruction set.

The general structure of a compiler is shown in figure 12, although many compilers might skip or combine two or more phases into one module. For more details on compiler design we refer to [6]. We clearly distinguish several intermediate representations stating that a compiler has all considerable knowledge of the internals of a program. While this knowledge is used for the translation process it is also processed to apply several code optimization techniques. A compiler for example performs some data and control flow analysis in order to build up an interference graph for the register allocation. This flow analysis could also be useful for code obfuscation itself.

Figure 12: The general structure of a compiler. Current compiler designs might skip or combine several phases.

In essence, code obfuscation has many similarities with code compilation. Both translate program code into other code while preserving its functionality. While compilation might have as one of its priorities code optimization, obfuscation focuses rather on obscurity in the first place. Of course there consist trade-offs. Obfuscation is not suitable if it requires an extreme amount of resources or if it makes a program slow and bulky.
If we focus more on the general scheme of a compiler, we examine where we could transform code resulting in more obscurity or diversity. This obscurity and diversity will obstruct code analysis and code decompilation.

A possible phase to introduce diversity for example is instruction selection. Instruction selection is actually mapping expression trees to machine instructions. This process is called “tiling” because many tile combinations are covering the expression tree. The algorithms that map tree patterns to instructions are optimized to produce mappings with a cost as low as possible, for example the one resulting in fewest instructions. The other tile combinations also translate our program into instructions, but this might be more. In this way we could translate a program into several other programs, which look a bit different at instruction level, but actually perform the same computations, as shown in figure 13. Whether these resulting programs are harder to understand or more obscure than the original one will determine if this mapping is also useful for obfuscation purposes. Software metrics or heuristics could estimate how much both instances actually differ and how much extra effort an attacker has to do to reverse-engineer one of them. By this example we show that a compiler itself is actually useful as an tool for introducing code diversity and applying code transformations.

5.2.3 Low Level Transformations

Similar to obfuscation at a source level, we could perform transformations on a lower level, for example object code, assembly or machine code. These transformations will thus be done after code compilation and we could therefore classify them as post compilation transformations.

For example, “ObjObf” [45] is a tool to obfuscate object files. Although, the tool is still under development it can perform several actions on a basic block level making it a lot harder for humans and programs to disassemble the resulting object files. ObjObf was based on extensive work from Wroblewski [56] who examined the potency of assembly code obfuscation.

5.3 Who will apply transformations and why

This part will focus on who can apply these transformations. Many companies want to deliver a ready-to-use software package to their users. In most cases this will be ready-to-install code. But in some cases (and also in the open source community) “ready-to-compile” code is distributed. A user who obtains this code just has to compile this source code resulting in an executable application. In both cases owners of software might want to protect their software or at least a critical part (an algorithm, a secret key
or sensitive embedded data) so that some security measures are needed. Obfuscation is therefore one of the recommended techniques to protect software in the scenario just mentioned.

**Scenario 1: obfuscation by the owner** Although, “security through obscurity” is claimed to be a bad strategy from a security point of view, this statement has to be read with caution. For example, a programmer might want to hide the internal working of his software containing certain flaws in the design. As obfuscation does not change the functionality, this program will still contain its weaknesses after obfuscation, although, it might be harder to identify and locate those weaknesses. Furthermore, obfuscation does not obstruct secure software development. Software development still has to avoid security flaws (e.g. buffer overflows) because transforming your code in another form gives no guarantee at all that all flaws will be eliminated. Perhaps in the worst case, more flaws will be created when a certain (unsave) transformation is applied on a certain critical module.

Although, the main aim from code obfuscation might be to obscure the
internal of a software program, in practice this translates itself in increasing complexity of the code to counter code analysis techniques. An obfuscator will never guaranty optimal protection. It rather will satisfy a security threshold specified by the developer to resist software threats for a specified period of time. After this period the software might become insecure and the software should be updated.

Scenario 2: obfuscation by the user Expanding the view that only software companies who develop and distribute software might be interested in applying code transformations, consider the following scenario. All users download an application from the Internet. One month later a virus is exploiting a recently reported bug in the software. Because most users did not install the patch yet, the virus spreads fast. Some users, however, have obfuscated their software themselves resulting in “personalized” versions which the virus fails to exploit. In this case we are talking about “security through diversity”.

Another application domain concerns using open source software. Here the source code is available. It seems that no internals need to be obscured or hidden as the source is public. Nonetheless, the disadvantage of the public source code is that attackers can easily download the source, analyze it, locate a flaw and write an automated tool to exploit this flaw. Users can
individualize or randomize their software by obfuscating the source after
download and before compiling it or by transforming the compiled code and
running their own obfuscated version. In this way they can protect their
software against automated attacks which exploit certain static properties
in the code. This method of diversifying software obstructs the spread of
viruses and other malicious software while the user still has full control over
its software and the original source code of the application. Note that this
ideology is already in use to secure software against buffer overflow attacks [?].

So to conclude this part, we described two distinct cases. In the first case
the software owner applies obfuscation in order to protect their software or
embedded information. In the second case it is the user who will obfuscate
obtained software in order to create a personalized version that is more
resistant to automated attacks exploiting commonly used software.
6 Applications

Software obfuscation has many application domains. The bigger the need for security and diversity becomes, the more code transformations and other techniques will be utilized to protect critical data and code in applications. Software will be more and more integrated in the application and more often critical code has to be protected against white box attacks.

6.1 E-commerce and E-banking Applications

Techniques such as obfuscation is mainly designed to facilitate securing of e-commerce and e-banking applications, which often embed cryptographic keys and critical information. In general, it can be used to protect all distributed client software where an owner looses control or where the user wants to protect against automated attacks.

6.2 Cryptographic Software

Current existing implementations of algorithms, e.g. encryption algorithms, can resist more global attacks through obfuscating code-transformations. Secret or sensitive data and private key information will be practically secured against runtime analysis. And by this, techniques such as time and power attacks will become increasingly complex and thus harder to deploy.

6.3 Digital Rights Management and Watermarking

A more commercial application of code obfuscation is Digital Rights Management (DRM), see also [40]. Because white-box cryptography and also obfuscation techniques are able to introduce a certain “randomness” in a software implementation, this implementation can be linked to the legitimate owner of that particular piece of software. Furthermore, the obfuscation techniques will avoid the software being decompiled and stolen or tampered with by a third party.

6.4 Embedded Systems and Smartcards

Design of secure software makes also more sense in embedded systems design once the computation and power restrictions disappear. With the evolving state of for example smartcards, it is feasible to run obfuscated software also on smaller embedded systems, such as micro controllers or smart cards. These systems are in particular very sensitive for white-box attacks because of their
high distribution and availability and their medium hardware protection. Examples of this are the credit cards where secret keys are embedded in the chip to perform cryptographic computation [12] ad hoc or pay television where decryption is done on the fly by a local hardware/software module at the receiver side.

6.5 Software Agents

In the case of software agents, the user of the agent actually becomes the owner of the agent. Once the agent is given a task it can travel through a network, which can be local, but also could be the Internet. The agent might carry valuable information from its owner that has to be protected during transmission, during temporarily storage and during execution action of the agents. Hence, presented software techniques could be easily applied on software agents.
7 Conclusions

In this project we explained the problem of software protection. We specified what type of attacks exist and stated why protection is necessary.

Furthermore, we summed several state of the art protection techniques which can be embedded in software to protect against analysis and tampering attacks. Although we summed all these possible techniques separately, it is of course possible to combine these techniques into one solution. Building an application that could transform a piece of software using combinations of these techniques just by clicks on a mouse will provide a practically strong security.

In another section we talked about code transformations, how to classify them and where to apply them during software development and enrollment. We also introduced who has advantages by using this techniques and linked this to code diversity.

Finally, we described several application domains where are techniques are most appropriate.
References

   http://www.codeboost.org/.

[2] Intro to spyware.  


