Self-encrypting Code to Protect Against Analysis and Tampering

Jan Cappaert, Nessim Kisserli, Dries Schellekens, and Bart Preneel

Katholieke Universiteit Leuven
Department of Electrical Engineering, ESAT/SCD-COSIC
Kasteelpark Arenberg 10
B-3001 Heverlee, Belgium
`{jcappaer,nkisserl,dschelle,preneel}@esat.kuleuven.be`

Abstract

Confidentiality and data authenticity are two basic concepts in security. The first guarantees secrecy of a message, while the latter protects its integrity. This paper examines the use of encryption to secure software static analysis and tampering attacks. We present the concept of code encryption, which offers confidentiality, and a method to create code dependencies that implicitly protect integrity. For the latter we propose several dependency schemes based on a static call graph which allow runtime code decryption simultaneous with code verification. If code is modified statically or dynamically, it will result in incorrect decryption of other code, producing a corrupted executable.

1 Introduction

From the early 60s until 80s application security was merely solved by designing secure hardware, such as ATM terminals, or set-top boxes. Since the 90s, however, secure software gained much interest due to its low cost and flexibility. Nowadays, we are surrounded by software applications which we use for webbanking, communication, e-voting, ... As a side effect, more threats such as piracy, reverse engineering and tampering emerge. These threats try to exploit critical and poorly protected software. This illustrates the importance of thorough threat analysis (e.g. STRIDE [13]) and new software protection schemes, needed to protect software from analysis and tampering attacks. This paper provides a technique to protect against the last two threats, namely reverse engineering and tampering.

For decades encryption has provided the means to hide information. Originally, it served for encrypting letters or communications, but quickly became a technique to secure all critical data, either for short-term transmission or long-term storage. While software enterprises offer commercial tools for performing software protection, an arms race is going on between the software programmers and the people attacking software. Although, encryption is one of the best understood information hiding techniques, encryption of software is still an open research area. In this paper we examine the use of self-encrypting code as a means of software protection.

Section 1 introduces our motivation and Section 2 describes software security and its threats. Section 3 gives a brief overview of related research, while Section 4 elaborates on code encryption. In Section 5 we present a framework that facilitates generation of tamper-resistant programs and show some empirical results as a proof of concept. Section 6 explains several attacks and possible countermeasures. And finally, Section 7 summarises this paper and outlines some conclusions.

2 Software security and threats

One of a company's biggest concerns is that their software falls prey to reverse engineering. A secret algorithm that is extracted and reused by a competitor can have major consequences for software companies. Also secret keys, confidential data or security related code are not intended to be analysed, extracted and stolen or even corrupted. Even if legal actions such as patenting and cyber crime laws are in place, reverse engineering remains a considerable threat to software developers and security experts.

Often the software is not only analysed, but also tampered with. In a branch jamming attack, for example, an attacker just replaces a conditional jump by an unconditional one, forcing a specific branch to execute even when it is not supposed to under those conditions. Such attacks can have a major impact on applications which involve licensing, billing, or even e-voting.

Before actually changing the code in a meaningful way, one always needs to understand the internals of a program. Changing a program at random places can no longer guarantee the correct working of the application after modification. Several papers present the idea of *self-verifying code* [2, 12] which is able to detect any changes to critical code. These schemes, however, do not protect against analysis of code. In this paper we try to solve analysis and tampering attacks simultaneously.

We can distinguish two main categories of analysis techniques: static analysis and dynamic analysis. *Static analysis* is applied to non-executing code, e.g. disassembly or decompilation [5]. *Dynamic analysis* is performed while the code is executed. It is typically easier to obstruct static analysis than protect the code against dynamic attacks.

In this paper we focus on software-only solutions because of their low cost and flexibility. It is clear that code encryption is useful if encrypted code can be sent to a secure co-processor [22]. But when this component is not available, as it is in most current systems, it becomes less obvious how to tackle this problem. As opposed to a black-box system, where the attacker is only able to monitor I/O of a process, an environment where the attacker has full privileges behaves like a white box, where everything can be monitored. Chow *et al.* [4] call this a *white-box environment* and propose a method to hide a key within an encryption algorithm.

3 Related research

There are three major threats to software: piracy, reverse engineering and tampering. Collberg *et al.* [9] give a compact overview of techniques to protect against these threats. Software watermarking for example aims at protecting software reactively against piracy. It generally embeds hidden, unique information into an application such that it can be proved that a certain software instance belongs to a certain individual or company. When this information is unique for each instance, one can trace copied software to the source unless the watermark is destroyed. The second group, code obfuscation, protects against reverse engineering. This technique consists of one or more program transformations that transform a program in such a way that its functionality remains the same but analysing the internals of the program becomes very hard. A third group of techniques aims to make software 'tamper-proof', also called tamper-resistant. As this paper investigates protection mechanisms against malicious analysis and tampering, we will not elaborate on software watermarking.

3.1 Code obfuscation

As software gets distributed worldwide, it becomes harder and harder to control it from a distance. This means that attackers often can analyse, copy, and change it at will. Companies however have been inventing techniques to make this analysis harder. The techniques range from small tricks to counter debugging, such as code stripping, to complex control flow and data flow transformations that try to hide a program's internals. This hiding tries to address the security objective of *confidentiality*. For example, when Java bytecode was shown to be susceptible to decompilation – yielding the original source code – researchers began investigating techniques to protect against this [7, 8, 15]. Protection of low-level code against reverse engineering has been addressed as well [25, 19].

3.2 Self-modifying code

While code obfuscation aims to protect code against both static and dynamic analysis, there exists another technique to protect against code analysis, namely self-modifying code. This technique offers the possibility to generate code at runtime, instead of transforming it statically. In practice however, self-modifying code is largely limited to the realm of viruses and malware. Nevertheless, some publications consider self-modifying code as a technique to protect against static and dynamic analysis. Madou *et al.* [16] for example consider dynamic code generation. They propose a technique where functions are constructed prior to their first call at runtime. Furthermore, clustering is proposed such that a common template can be used to construct each function in a cluster, performing a minimal amount of changes. To protect the constant 'edits' against dynamic analysis, the authors propose use of a pseudo random number generator (PRNG).

Our decryption at runtime technique is equivalent with code generation, except the fact that the decryption key can rely on other code, rather then on a PRNG. Furthermore minimises re-encryption the visability of code during execution, while Madou *et al.* do not explicitly protect a function template after the function executed.

3.3 Tamper resistance

Protecting code against tampering can be considered as the problem of *data authenticity*, where in this context 'data' refers to the program code. In '96 Aucsmith [1] illustrated in his paper a scheme to implement tamper-resistant software. His technique protects against analysis and tampering. For this, he uses small, armoured code segments, also called integrity verification kernels (IVKs), to verify code integrity. These IVKs are protected through encryption and digital signatures such that it is hard to modify them. Furthermore, these IVKs can communicate with each other and across applications through an integrity verification protocol. Many papers in the field of tamper resistance base their techniques on one or more of Aucsmith's ideas.

Several years later, Chang *et al.* [2] proposed a scheme based on *software guards*. Their protection scheme relies on a complex network of software guards which mutually verify each other's integrity and that of the program's critical sections. A software guard is defined as a small piece of code performing a specific tasks, e.g. checksumming or repairing. When checksumming code detects a modification, repair code is able to undo this malicious tamper attempt. The security of the scheme relies partially on hiding the obfuscated guard code and the complexity of the guard network. A year later, Horne *et al.* [12] elaborated on the same idea and proposed 'testers', small hashing functions that verify the program at runtime. These testers can be combined with embedded software watermarks to result in a unique, watermarked, self-checking program. Other related research is *oblivious hashing* [3] which interweaves hashing instructions with program instructions and which is able to prove whether a program operated correctly or not.

Recently, Ge *et al.* [10] published a paper on control flow based obfuscation. Although the authors published their work as a contribution to obfuscation, the control flow information is protected with an Aucsmith-like tamper resistance scheme.

4 Code encryption

The following sections give an overview of dynamic code encryption. This is encrypting binary code at runtime. Often this is also covered by the terms self-modifying or self-generating code. Encryption generally assures the confidentiality of the data. In the context of binary code, this technique mainly protects against static analysis. For example, several encryption techniques are used by polymorphic viruses [21] and polymorphic shell code [6]. In this way, they are able to bypass intrusion detection systems, virus scanners, and other pattern-matching interception tools. The following sections present several methods of encrypting code at runtime.

4.1 Bulk encryption

If a program is encrypted completely with a single routine, we call it *bulk encryption.* The decryption routine is usually prepended to the encrypted body. At runtime this routine decrypts the body and then transfers control to it. The decrypting routine can either consult an embedded key or fetch one dynamically (e.g. from user input or from the operating system). The main advantage of such a mechanism is that as long as the program is encrypted its internals are hidden and therefore protected against static analysis. Another advantage is that the encrypted body makes it hard for an attacker to statically change bits in a meaningful way. Changing a single bit will result in one or more bit flips in the decrypted code and thus modifying one or more instructions, which might crash the program or cause other unintended behaviour due to binary code's brittleness. Nevertheless, a simple construction such as bulk encryption, has certain desirable properties:

- it protects the code against static analysis and forces an attacker to perform a dynamic attack;
- as long as the code is encrypted, it is protected against targeted tampering;
- it has a very limited overhead in size and performance as the encryption is done all at once.

However, as all code is decrypted simultaneously, it is inherently weak. An attacker simply waits for the decryption to occur before dumping the process image to disk in clear form for analysis.

4.2 Partial encryption

In contrast to bulk encryption where program code is decrypted all at once, one could increase granularity and decrypt small parts at runtime. Shiva [17] is a binary encryptor that uses obfuscation, anti-debugging techniques and multi-layer encryption to protect

ELF binaries. However, to the best of our knowledge it still encrypts large code blocks, although one at a time, and thus exposes large portions of code at runtime. Viega *et al.* [24] provide a method in C to write self-modifying programs that decrypt a function at runtime. While implementing self-modifying code on a high level is not straightforward (no address information is known before compilation), their proposed solution is easy-to-use as it is based on predefined macros, an external encryption program and a four step build phase, which goes as follows:

- initial build: the code is instrumented to collect the required address information;
- the actual address information is generated by executing the instrumented executable;
- final build: software is built and the necessary encryption routines are put in place;
- an external encryption program uses the address information to encrypt the function that should be initially hidden.

Figure 1 illustrates how a function will be decrypted at runtime. A function `cipher` is used to modify (decrypt) a block `code`. The code block `key` is read and used as key.

This scheme overcomes the weaknesses of revealing all code at runtime as it offers the possibility to decrypt only the necessary parts, instead of the whole body, as bulk encryption usually does. The disadvantage is a slight increase in overhead due to multiple calls to the decryption routine.

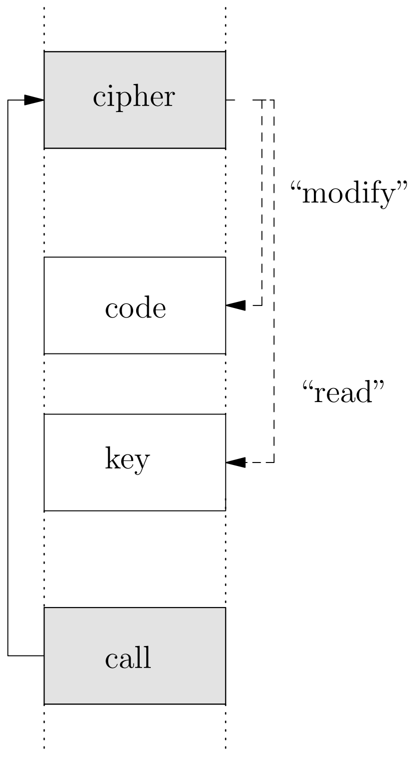


Figure 1: A basic scheme for function decryption where correct decryption of a function, called `code`, depends on another function's code, called `key`. The code that performs this operation is called `cipher`.

5 Function encryption framework

5.1 Basic principle

For our code encryption framework we rely on function encryption and code dependencies. For this we use the principle of encrypting functions mentioned above. We define a new kind of *software guard*, which decrypts (D) or encrypts (E) the code of a function a using the code of another function b. Using parameters a and b decryption can be expressed as $a = D_b(A)$, where A is the encrypted function a or $A = E_b(a)$. Furthermore, we would like the guard to have the following properties:

- if one bit is modified in b, then 1 or more bits in a should change; and
- if one bit is modified in A, then 1 or more bits should be modified in a after decryption.

Many functions meet these requirements however. For the first requirement a cryptographic function with b as key could be used. For example Viega *et al.* [24] use the stream cipher RC4 where the key is the code of another function. The advantage of an additive stream cipher is that encryption and decryption are the same computation, thus the same code. This also holds for certain block ciphers, such as DES [18], but not for all (e.g. AES); using a suitable mode of operation, like counter mode (in this case you use the block cipher as a stream cipher), can overcome this inconvience. However, the key size for symmetric cryptographic algorithms is limited, e.g. to 128 or 256 bits. RC4 for example allows a key up to 256 bytes. This means that any modification to b beyond the first 256 bytes will not cause any change to a, which violates our first requirement. Therefore we need some kind of compression function that maps a variable length code block to a fixed length string, which is then used as key for the encryption routine. Possible functions are checksum functions, e.g. CRC32, or cryptographic hash functions [18].

Using code of b to decrypt A could be seen as an implicit way of creating tamper resistance; modifying b will result in an incorrect hash value, i.e. encryption key, and consequently incorrect decryption of A. Furthermore, flipping a bit in A will flip at least one bit in a; in case of an additive stream cipher a bit change in the ciphertext will happen at the same location in the plaintext, while the error propagation for block ciphers depends on the used mode of operation. This might be sufficient to make most applications crash due to binary code's brittleness. For example, a single bit flip in the clear code might change the opcode of an instruction, resulting in an incorrect instruction to be executed, but also in desynchronising the next instructions [14], which most likely will crash the program. Changing one of the operands of an instruction will cause incorrect or unpredictable program behaviour.

Another advantage of this scheme is that the key is computed at runtime (relying on other code), which means the key is not hard-coded in the binary and therefore hard to find through static analysis (e.g. entropy scanning [20]). The main disadvantage is performance: loading a fixed length cryptographic key is usually more compact and faster than computing one at runtime, which may involve calculating a hash value. Furthermore, the key setup of symmetric cryptographic algorithm will also have a performance impact.

Although we believe that cryptographic hash functions and stream ciphers are more secure, we used for our experiments a self-designed XOR-based scheme – which satisfies our two properties – to minimise the cost in speed and size after embedding the software guards.

5.2 Dependency schemes

With this basic function encryption method we now can build a network of code dependencies that make it hard to change code statically or dynamically. We propose three schemes which are based on call graph information to make functions depend on each other such that static and dynamic tampering becomes difficult.

Scheme 1 Initially all callees, the called functions, are encrypted, except `main()`, which has to be in clear when the program transfers control to it. A function is decrypted before its call and the decryption key is based on the code of the caller, the calling function.

Note that in the above case once a function is decrypted it stays so and is susceptible to static analysis, e.g. if a user forces a dump of the process' memory space.

Scheme 2 Initially all callees are encrypted. Their caller calls a guard to decrypt them just before they are called and to re-encrypt them when they return. Again the decryption key is based on the code of the calling function. This will tampering of the caller without affecting the callees very difficult.

We remark that if a function is only decrypted before it is called and is re-encrypted after it returns, then the code of all callers in the call path (this is the path in the call graph leading to the called function) will be in cleartext.

Scheme 3 Initially all callees are encrypted. Each caller decrypts its callee before the call and re-encrypts it after it returns. Additionally, the callee encrypts its caller upon being called and decrypts it before returning.

In this last case the maximum number of functions in cleartext during execution is minimised. Though, guard code is implicitly considered to be in cleartext as well.

The memory layout of a function call protected according to scheme 3 is sketched in Figure 2. A function is called through the following steps:

1. a guard is called to decrypt the callee;

2. control is transferred to the callee;

3. the callee calls a guard to encrypts his caller;

4. the callee executes;

5. before returning, the callee calls a guard to decrypt the caller code;

6. control is transferred to the caller;

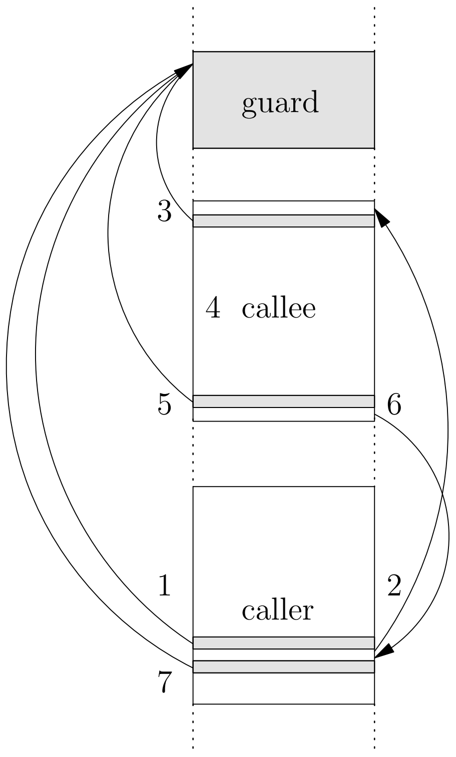


Figure 2: Memory layout of scheme 3: 1, 3, 5, and 7 are guard calls; 2 and 6 are control transfers.

7. the caller calls a guard to re-encrypt the callee code.

All functions that call a guard to decrypt or encrypt another function use their own code as key material. It can be shown that in this case tampering will always be detected ('detected' here implies that incorrect execution and undesired behaviour will appear):

- If a function is tampered with while it is encrypted, this will result in a modified version and all callees of this function will be decrypted incorrectly (and their callees as well, etc.). This is $\bar{a} = D_b(\bar{A})$.

- If a function is tampered with while it is decrypted, this will result in incorrect decryption of the callees that are decrypted after this moment in time. This corresponds to $\bar{a} = D_{\bar{b}}(A)$.

Note that it is also possible to generate schemes based on heuristic information, such as other software guards [2] do, where the owner specifies which code is critical and where the protection techniques focus on that part only. However, our schemes are applied to the whole call graph, thus protecting the whole binary.

Consider a program P and its modified version $\bar{P}$, then we define the *time cost* C_t and the *space cost* C_s as

$$C_t(P, \bar{P}) = \frac{T(\bar{P})}{T(P)}$$
$$C_s(P, \bar{P}) = \frac{S(\bar{P})}{S(P)}$$

where $T(X)$ is the execution time of program X and $S(X)$ its size. Table 1 gives an overview the performance cost of schemes 1, 2, and 3 applied on some basic implementation of common UNIX commands. The command `du` outputs how much space files represent on disk, `tar` is an archiving utility, and `wc` is a program that counts words in a file. We clearly notice that `wc` has the largest performance loss when protected by scheme 3. This is due to the numerous loops that also contain a lot of guard calls to decrypt or re-encrypt code. Calling more guards outside the loops would speed up the program, but it would reveal functions longer than needed at runtime. Scheme 1 appears to run faster than the original program. This might be a result of extensive caching of code and data fragments (our code is treated as data when we decrypt it). Space cost ranged from 1.031 to 1.170 which is an increase of 3 up to 17% of the original program size. This expansion is proportional to the number of guard calls and the size of the guard code.

Program	Scheme 1	Scheme 2	Scheme 3
`du`	0.899	3.612	8.364
`tar`	0.822	1.339	2.783
`wc`	0.989	39.715	91.031

Table 1: Performance cost C_t when using self-modifying code with dependency schemes.

5.3 Scheme restrictions

Opposed to the simplicity of our schemes, generic programs face each scheme to specific difficulties. Some of them are easy, others harder to solve. An overview is given below.

Loops Scheme 1 poses a problem when a function call is nested within a loop and is decrypted prior the function call; during different iterations the function will be decrypted multiple times resulting in incorrect code. Therefore, we propose placing the decryption routine outside the loop. Schemes 2 and 3 do not encounter this problem as they always re-encrypt the called function after it returns. However, placing the encryption and decryption routines outside the loop could always be considered for performance reasons. This implies code will be decrypted and thus unprotected as long as the loop is running, but it reduces overhead.

Recursion For all three schemes care with recursion needs to be taken. If a function calls itself (pure recursive call), it should – according to our scheme definitions – decrypt itself, although it is in cleartext already. Therefore, we decrypt a recursive function only once: namely when it gets called by another function. We can extend this to recursive cycles, where a group of functions together form a recursion.

Multiple callers If a function a is called by different callers b_i, one could choose to encrypt the callee a with the cleartext code of only one of the callers, e.g. based on profiling information. The function that calls the particular callee the most, could then be used as key to decrypt it. However, when another caller is modified, this will not result in incorrect decryption of a. Therefore we state that the decryption of A should rely on all b_i. The problem is: when a is called, only one b_i might be in clear. To encrypt all b_i yields a number of guard calls that decrypt the paths from the actual caller to the key code functions. After this, it should actually re-encrypt all the decrypted functions to reduce visibility in memory. The maximum number of decryptions required to get the key code in cleartext for a pair of callers b_x and b_y is $l_x + l_y$ where l_x and l_y are the nesting levels of a_x, respectively a_y, relative to `main()`. The same number of encryptions is needed to re-encrypt all these functions after the target function is decrypted. In the case of n callers, we need $\sum_{i=1}^{n} l_i$ guards in the worst case to decrypt all callers b_i and then another n guards to decrypt A. To overcome this overhead we propose to rely on the encrypted code of the callers, namely all B_i, and decrypt A even before control is given to any b_i. For this we only need n extra guards instead of $n + \sum_{i=1}^{n} l_i$ and any change in any caller will still be propagated to the callee.

Other options involve:

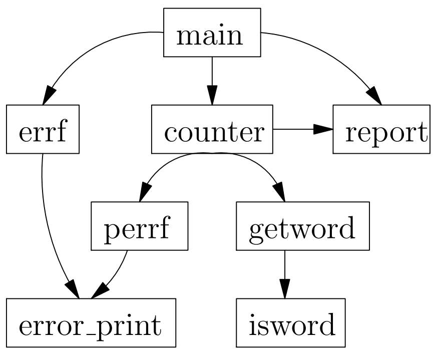
- not encrypting functions which have multiple callers (most trivial solution), which violates our schemes;
- inlining the callee, but the callee may call other functions itself, which only shifts the problem, because the callees of the inlined callee will have multiple callers after inlining;
- encrypting with another (possibly encrypted) code as key code, e.g. the encryption code itself (see also Section 6.1);
- modifying the guard, such that it foresees a correction value c which compensates the hash of a function b_2 when function a is encrypted with b_1 yielding $hash(b_1) = hash(b_2) \oplus c$. However, this value also facilitates attackers to modify code. All they have to do is compute the new hash and compensate it in the correction value.

As an example we refer to Figure 3 where `error_print()` can be called by `errf()` and `perrf()`. Relying on their clear code would yield decryption of `main()` as well as `counter()` to decrypt `error_print()`. When relying however on encrypted code, we can decrypt `error_print()` just before one of its callers is called and rely on their encrypted code.

5.4 Code encryption as an addition to code verification

Our guards, which modify code depending on other code, offer several advantages over the software guards proposed by Chang *et al.* [2] that only verify (or repair) code:

- confidentiality: as long as code remains encrypted in memory it is protected against analysis attacks. With a good code dependency scheme it is feasible to ensure only a minimal number of code blocks are present in memory in decrypted form;
- tamper resistance: together with a good dependency scheme, our guards offer protection against any tampering attempt. If a function is tampered with statically or even dynamically, the program will generate corrupted code when this function is executed and will most likely eventually crash due to illegal instructions. Furthermore, if the modification generates executable code, this change will be propa-



```
1     main {wc.c 124}
2          errf {wc.c 32}
4               error_print {wc.c 20}
10         counter {wc.c 105}
12              perrf {wc.c 44}
14                   error_print ... {4}
16              getword {wc.c 75}
19                   isword {wc.c 62}
23              report {wc.c 55}
25         report ... {23}
```

Figure 3: Static call graph and tree of the UNIX word count command `wc`. The reduced static call tree was produced with GNU's cflow [11].

gated to other functions, resulting in erroneous code.

In some cases, programmers might opt for self-checking code instead of self-encrypting code, based on some of the following disadvantages:

- implicit reaction to tampering: if a verified code section is tampered with the program will crash (if executed parts rely on this modified section). However, crashing is not very user-friendly. In the case of software guards [2, 12], detection of tampering could be handled more technically by triggering another routine that for example exits the program after a random time, calls repair code that fixes the modified code (or a hybrid scheme, which involves both techniques), ...

- limited hardware support: self-modifying code requires memory pages to be executable and writable at the same time. However some operating systems enforce a WˆX policy as a mechanism to make the exploitation of security vulnerabilities more difficult. This means a memory page is either writable (data) or executable (code), but not both. Depending on the operating system, different approaches exist to bypass – legally – the WˆX protection: using `mprotect()`, the system call for modifying the flags of a memory page, to explicitly mark memory readable and executable (e.g. used by OpenBSD) or setting a special flag in the binary (e.g. in case of PaX). A bypass mechanism will most likely always exist to allow for some special software like a JVM that optimises the translation of Java bytecode to native code on the fly.

6 Attacks and improvements

6.1 inlining of guard code

If implementation of dependency schemes (see also Section 4.2) consists of a single instance of the guard code and numerous calls to it, an attacker can modify the guard or crypto code to write all decrypted content to another file or memory region. To avoid that an attacker only needs to attack this single instance of the guard code, inlining the entire guard could thwart this attack and force an attacker to modify all instances of the guard code at runtime, as all nested guard code will initially be encrypted. However, a disadvantage of this

Program	Scheme 1	Scheme 2	Scheme 3
du	1.088	1.379	1.753
tar	1.213	1.484	2.219
wc	0.458	2.210	2.800

Table 2: Space cost C_s when using self-modifying code with dependency schemes after inlining all guards.

inlining is code expansion. Compact encryption routines might keep the spacial cost relatively low, but implementations of secure cryptographic functions are not always small.

Table 2 shows that `wc` almost tripled in size after inlining guards according to scheme 3. The performance cost, however, jumped from 91.031 to 1379.71 which was our worst case result after inlining guards. The program `tar` ran only 35 times slower after inlining guards as specified by scheme 3. Further optimisation of the guard code, and the cryptographic algorithms should contribute to a lower space cost and as a consequence also a smaller performance penalty, as some guard code has to be decrypted by other guards and so on.

6.2 Hardware-assisted circumvention attack

Last year, van Oorschot *et al.* [23] published a hardware-assisted attack that circumvents self-verifying code mechanisms. The attack exploits differences between data reads and instruction fetches. This is feasible due to the fact that current computer architectures distinguish between data and code. When instructions are verified (e.g. checksummed or hashed) they are treated as data, but when instructions are fetched for execution they are treated as code. The attack consists of duplicating each memory page, one page containing the original code, while another contains tampered code. A modified kernel intercepts every data read and redirects it to the page containing the original code, while the code that gets executed is the modified one.

Our protection scheme is different however. Redirecting the data reads to a page with unmodified code will result in a correct hash to decrypt the next function. However, this attack implies that code is in clear and thus can be modified. The only blocks in cleartext however are the guard code and `main()` and they can thus be modified using van Oorschot's attack. If the decryption routines are not inlined, then an attacker could just modify the encryption code (e.g. to redirect all generated cleartext at runtime), while – if the integrity of the decryption routine is verified – this will not be detected due to the redirection of the data reads. When cryptographic routines are inlined, extending this attack to the inlined decryption routines would require to duplicate every memory page, as soon as a function gets decrypted, and modify its decrypted body dynamically. This whole attack implies intercepting 'data writes' that modify code, use this as a trigger to dynamically copy a memory page, and modify code dynamically. This attack however is identical to a dynamic analysis attack, where functions get decrypted, decryption routines are identified and all clear code is intercepted, allowing an attacker to rebuild the application without protection code and modifying it afterwards statically.

6.3 Increasing granularity

Our scheme is built on top of static call graph information and therefore uses functions as building blocks. Our implementation also uses function pointers, which can be addressed at a high level (e.g. C). Implementing self-verifying or self-modifying code however can work with any granularity if implemented carefully. The only rule that should be respected is: code should be in cleartext form (correct binary code instructions, part of the original program) whenever it is executed.

With inline Assembly (`asm()` in `gcc`) we can inline Assembly labels in the C code. Just as function pointers their scope is global. However, these labels can be placed anywhere in a function, unlike function pointers which by definition only occur at the beginning of the function and which must be defined before they can be used. A further benefit of using labels is the elimination of the initial build phase which gathered address information. Providing the right addresses to the guard code is in this case done by the assembler which just replaces the labels by the corresponding addresses.

If one increases the granularity, and encrypts parts of functions, the guards can be integrated into the program's control flow which will make it even harder to analyse the network of guards, especially when they are inlined. However, we believe that such a fine-grained structure will induce much more overhead. The code blocks to be encrypted will be much smaller than the added code. Furthermore, more guards will be required to cover the whole program code. Hence it is important to trade-off the use of these guards and perhaps focus on some critical parts of the program and avoid 'hot spots' such as frequently executed code.

7 Conclusions

This paper presents a new type of software guards which are able to encipher code at runtime, relying on other code as key information. This technique offers confidentiality of code, a property that previously proposed software guards [2, 12] did not offer yet. As code is used as a key to decrypt other code, it becomes possible to create code dependencies which make the program more tamper-resistant. We therefore propose three dependency schemes, which are built on static call graph information. These schemes make sure an introduced modification is propagated through the rest of the program, forcing the application to work incorrectly or exit prematurely. As a proof of concept we implemented our technique in C and applied it on some small C programs.

Acknowledgements

This work was supported in part by the Research Foundation - Flanders (FWO), the Institue for the Promotion of Innovation through Science and Technology in Flanders (IWT), the Interdisciplinary Institute for BroadBand Technology (IBBT), and the Concerted Research Action (GOA) Ambiorics 2005/11 of the Flemish Government.

References

[1] D. Aucsmith. Tamper resistant software: an implementation. *Information Hiding, Lecture Notes in Computer Science*, 1174:317–333, 1996.

[2] H. Chang and M. J. Atallah. Protecting software codes by guards. *ACM Workshop on Digital Rights Managment (DRM 2001)*, LNCS 2320:160–175, 2001.

[3] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. Jakubowski. Oblivious hashing: a stealthy software integrity verification primitive. In *Information Hiding*, 2002.

[4] S. Chow, P. Eisen, H. Johnson, and P. van Oorschot. A White-Box DES Implementation for DRM Applications. In *Proceedings of 2nd work ACM Workshop on Digital Rights Management (DRM 2002)*, November 18 2002.

[5] C. Cifuentes and K. Gough. Decompiling of binary programs. *Software – Practice & Experience*, 25(7):811–829, 1995.

[6] CLET team. Polymorphic shellcode engine using spectrum analysis. `http://www.phrack.org/phrack/61/p61-0x09_Polymorphic_Shellcode_Engine%.txt`.

[7] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report #148, Department of Computer Science, The University of Auckland, 1997.

[8] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. *Principles of Programming Languages 1998, POPL'98*, pages 184–196, 1998.

[9] C. S. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. In *IEEE Transactions on Software Engineering*, volume 28, pages 735–746, August 2002.

[10] J. Ge, S. Chaudhuri, and A. Tyagi. Control flow based obfuscation. In *DRM '05: Proceedings of the 5th ACM workshop on Digital rights management*, pages 83–92, 2005.

[11] GNU. GNU cflow. `http://www.gnu.org/software/cflow/`.

[12] B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance. In *Proceedings of Workshop on Security and Privacy in Digital Rights Management 2001*, pages 141–159, 2001.

[13] M. Howard and D. C. LeBlanc. *Writing Secure Code, Second Edition.* Microsoft Press, 2002.

[14] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, 2003.

[15] D. Low. Java Control Flow Obfuscation. Master's thesis, University of Auckland, New Zealand, 1998.

[16] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. In J. Song, T. Kwon, and M. Yung, editors, *The 6th International Workshop on Information Security Applications (WISA 2005)*, volume LNCS 3786, pages 194–206. Springer-Verlag, August 2006.

[17] N. Mehta and S. Clowes. Shiva – ELF Executable Encryptor. Secure Reality. `http://www.securereality.com.au/`.

[18] A. Menez, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Inc., 1997.

[19] Scut, Team Teso. Burneye – x86/Linux ELF Relocateable Object Obfuscator.

[20] A. Shamir and N. van Someren. Playing "Hide and Seek" with Stored Keys. *Financial Cryptography '99*, LNCS 1648:118–124, 1999.

[21] Symantec. Understanding and Managing Polymorphic Viruses. `http://www.symantec.com/avcenter/reference/striker.pdf`.

[22] J. D. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. In *IP Workshop Proceedings*, 1994.

[23] P. C. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, 02(2):82–92, 2005.

[24] J. Viega and M. Messier. *Secure Programming Cookbook for C and C++*. O'Reilly Media, Inc., 2003.

[25] G. Wroblewski. *General Method of Program Code Obfuscation.* PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.