Towards Tamper Resistant Code Encryption: Practice and Experience

Jan Cappaert[1], Bart Preneel[1],
Bertrand Anckaert[2], Matias Madou[2], and Koen De Bosschere[2]

[1] Katholieke Universiteit Leuven
Department of Electrical Engineering, ESAT/SCD-COSIC
Kasteelpark Arenberg 10
B-3001 Heverlee, Belgium
{jan.cappaert,bart.preneel}@esat.kuleuven.be
[2] Universiteit Gent
Department of Electronics and Information Systems, ELIS/PARIS
Sint-Pietersnieuwstraat 41
B-9000 Gent, Belgium
{banckaer,mmadou,kdb}@elis.ugent.be

Abstract. In recent years, many have suggested to apply encryption in the domain of software protection against malicious hosts. However, little information seems to be available on the implementation aspects or cost of the different schemes. This paper tries to fill the gap by presenting our experience with several encryption techniques: bulk encryption, an on-demand decryption scheme, and a combination of both techniques. Our scheme offers maximal protection against both static and dynamic code analysis and tampering. We validate our techniques by applying them on several benchmark programs of the CPU2006 Test Suite. And finally, we propose a heuristic which trades off security versus performance, resulting in a decrease of the runtime overhead.

1 Introduction

In the 1980s application security was achieved through secure hardware, such as ATM terminals, or set-top boxes. Since the 1990s, however, secure software has gained much interest due to its low cost and flexibility. Nowadays, we are surrounded by software applications for online banking, communication, e-voting, ... As a result, threats such as piracy, reverse engineering and tampering have emerged. These threats are exacerbated by poorly protected software. Therefore, it is important to have a thorough threat analysis (e.g., STRIDE [10]) as well as software protection schemes. The techniques discussed in this paper protect against reverse engineering and tampering.

The goal of encryption is to hide the content of information. Originally, it was applied within the context of communication, but has become a technique to secure all critical data, either for short-term transmission or long-term storage. More recently, commercial tools for software protection have become available.

These tools need to defend against attackers who are able to execute the software on an open architecture and thus, albeit indirectly, have access to all the information required for execution.

Even though encryption is one of the best understood information hiding techniques and has been used previously for software protection, little details on its practical application and performance impact are available. In this paper we discuss a number of practical schemes for self-encrypting code and report on our experience. Furthermore, we introduce an on-demand decryption scheme which operates at function granularity and which uses the hash of other code sections as the decryption key.

The remainder of this paper is structured as follows. Section 2 describes software security and its threats. Section 3 provides an overview of related work. In Sect. 4 we discuss our on-demand code decryption scheme. A numerical evaluation is given in Sect. 5. Section 6 discusses several attack scenarios and possible countermeasures. Finally, conclusions are drawn in Sect. 7.

2 Software Security and Threats

Protecting code from reverse engineering is one of the main concerns for software providers. If a competitor succeeds in extracting and reusing a proprietary algorithm, the consequences may be major. Furthermore, secret keys, confidential data or security related code are not intended to be analyzed, extracted, stolen or 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.

In many cases, the software is not only analyzed, but also tampered with. Nowadays examples are cracks for gaming software or DRM systems. In a branch jamming attack, for example, an attacker replaces a conditional jump by an unconditional one, forcing a specific branch to be taken even when it is not supposed to under those conditions. Such attacks could have a major impact on applications such as licensing, DRM, billing and voting.

Before changing the code in a meaningful way, one always needs to understand the internals of a program. If one would change a program at random places one could no longer guarantee the correct working of the application after modification. Several papers present the idea of *self-verifying code* [4,9] that is able to detect any changes to critical code. These schemes, however, do not protect against analysis of code. In this paper tries to solve analysis and tampering attacks simultaneously through encryption.

We focus on software-only solutions because of their low cost and flexibility. Clearly, code encryption is more powerful if encrypted code can be sent to a secure co-processor [21]. But when this component is not available, as is the case on most existing systems, the problem becomes harder to tackle. Essentially, such a co-processor can be assumed to be a black-box system, where the attacker is only able to monitor I/O behavior. Software-only solutions against malicious

hosts need to work within a white-box environment, where everything can be inspected and modified at will.

3 Related Work

There are three major threats to software: piracy, reverse engineering and tampering. In recent years, a number of countermeasures have been treated in the literature. Software watermarking, for example, aims at protecting software reactively against piracy. It embeds a unique identifier into an application such that it can be proved that a specific copy belongs to a specific individual or company. As a result, one can trace copied software to the source unless the watermark is destroyed.

A second countermeasure, code obfuscation, protects against reverse engineering. Code obfuscation aims to generate a semantically equivalent, but less intelligible version of a program.

The goal of a third countermeasure is to make software more tamper-resistant. As this paper studies protection mechanisms against malicious analysis and tampering, we will not elaborate on software watermarking.

3.1 Code Obfuscation

Once software is distributed, it is largely beyond the control of the software provider. This means that attackers can analyze, copy, and change it at will. Not surprisingly, a substantial amount of research has gone into making this analysis harder. The developed techniques range from tricks to counter debugging, such as code stripping, to complex control flow and data flow transformations that try to hide a program's internal operation. The goal is to achieve 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 the code [6,13]. Protection of low-level code against reverse engineering has been addressed as well [24].

3.2 Tamper Resistance

Tamper resistance protects *data authenticity* where, in this context, 'data' refers to the program code. In '96 Aucsmith [1] introduced a scheme to implement tamper-resistant software. Through small, armored code segments, referred to as integrity verification kernels (IVKs), the integrity of the code is verified. 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. [4] proposed a scheme based on *software guards*. Their protection scheme relies on a complex network of software guards

which can 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 task (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. Horne et al. [9] discuss a related concept called 'testers', small hash functions that verify the program at run time. These testers can be combined with embedded software watermarks to result in a unique, watermarked, self-checking program.

Other related research is *oblivious hashing* [5], which interweaves hashing instructions with program instructions and which is able to prove to some extent whether a program operated correctly or not.

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

- limited hardware support: self-modifying code requires memory pages to be executable and writable at the same time. However some operating systems enforce a W^X 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 W^X 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 such as a JVM that optimizes the translation of Java bytecode to native code on the fly.
- implicit reaction to tampering: if an encrypted code section is tampered with the program will crash after incorrect decryption, assuming that it is hard to target bits flips in the plaintext by manipulating the ciphertext. Furthermore, even if one succeeds in successful tampering with a specific function, our dependency scheme will propagate faulty decryption along the functions on the call stack whenever the modified function is verified (i.e. a decryption key is derived from its code), which will sooner or later make the program crash as well. However, crashing is not very user-friendly. In the case of software guards [4,9], 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), warns the owner about the malicious attempt through a hidden channel, ...

3.3 Code Encryption

This section provides an overview of dynamic code decryption and encryption; one often refers to this as a specific form of self-modifying or self-generating code. Encryption ensures 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 and polymorphic shell code [18]. In this way, they are able to bypass intrusion detection systems, virus scanners, and other pattern-matching interception tools.

Bulk Decryption. We refer to the technique of decrypting the entire program at once as *bulk decryption*. The decryption routine is usually added to the encrypted body and set as the entry point of the program. At run time 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 (depending on the encryption scheme) and thus one or more modified instructions, which may lead to program crashes or other unintended behavior due to the brittleness of binary code.

However, as all code is decrypted simultaneously, an attacker can simply wait for the decryption to occur before dumping the process image to disk.

On-demand Decryption. In contrast to bulk decryption, where the entire program is decrypted at once, one could increase granularity and decrypt small parts when they are needed at run time. Once they are no longer needed, they can be re-encrypted. This technique is applied, a.o., by Shiva [14], a binary encryptor that uses obfuscation, anti-debugging techniques, and multi-layer encryption to protect ELF binaries. Viega et al. [23] provide a related method to write self-modifying programs in C that decrypt a function at run time.

On-demand decryption overcomes the weaknesses of revealing all code in the clear at once as it offers the possibility to decrypt only the necessary parts, instead of the whole body. The disadvantage is an increase in overhead due to multiple calls to the decryption and encryption routines.

4 On-demand Decryption Framework

In this section, we introduce our on-demand decryption scheme. The granularity of this scheme is the function level, meaning that we will decrypt and encrypt an entire function at a time.

4.1 Basic principle

The scheme relies on two separate techniques, namely integrity checking and encryption. The techniques from integrity-checking are used to compute the keys for decryption and encryption. The integrity checking function can be a

checksum function or a hash function. Essentially, it has to map a vector of bytes, code in this case, to a fixed-length value, in such a way that it is hard to produce a second image resulting in the same hash.

The basic idea is to apply the integrity checking function h to a function a to obtain the key for the decryption of another function b. Using the notation of D for decryption and E for encryption, this results in $b = D_{h(a)}(E_{h(a)}(b))$. To this end, b needs to have been encrypted with the correct key on beforehand (i.e. $E_{h(a)}(b)$, denoted by $\bar{b}$). We will refer to this scheme as a *crypto guard*.

We would like the guard to have at least the following properties:

- if one bit is modified in a, then 1 or more bits in b should change (after decryption); and
- if one bit is modified in $\bar{b}$, then 1 or more bits should change in b after decryption.

For the first requirement, a cryptographic function with a as key could be used. For example, Viega et al. [23] use the stream cipher RC4 where the key is code of another function. The advantage of an additive stream cipher is that encryption and decryption are the same computation, thus the same code. It is also possible to construct stream ciphers out of block ciphers (e.g., AES) using a suitable mode of operation. For more on the cryptographic properties of these functions, we refer to [15]. The major disadvantage of these ciphers is that they are relatively slow for our case, and relatively large as well. However, note that we still require the integrity-checking function that also serves as a one-way compression function, because each cipher requires a fixed, limited key size.

From a cryptographic point of view we require a second image resistant hash function and secure encryption mode with suitable error propagation properties (e.g., PCBC). However, size and speed of these algorithms is essential for the overall performance of the protection scheme as its security assumes inlining the guard code. This results in more code to be hashed and decrypted, and thus a higher cost. It is also possible to link the hashing itself to other code by using a keyed hash function, such as HMAC. Other proposals for hash-like functions and encryption routines are constructions based on T-functions, introduced by Shamir et al. [11]. These light-weight functions are popular as they have a direct equivalent available in both software and hardware. Nevertheless, it is still unclear whether constructions based on T-function are cryptographically secure.

As software tamper resistance is typically defined as techniques that make tampering with code harder, we can illustrate that our crypto guards offer protection against tampering. Namely, using code of a to decrypt (i.e. deriving a decryption key from a's code) could be seen as an implicit way of creating tamper resistance; modifying a will result in an incorrect hash value (i.e. encryption key), and consequently incorrect decryption of $\bar{b}$.

Furthermore, changing $\bar{b}$ will result in one or more changes to b; in case of an additive stream cipher a bit change in the ciphertext will correspond to a bit change the plaintext at the same location. However, if this plaintext itself is used as key material in a later stage (e.g, to derive decryption keys for its callees), this

will result in incorrect code. Furthermore, due to the brittleness of binary code and the denseness of the IA32 instruction set, 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 desynchronizing the next instructions [12], which most likely will lead to a crashing program.

Another advantage of this scheme is that the key is computed at run time, which means the key is not hard-coded in the binary and therefore hard to find through static analysis (e.g., entropy scanning [17]). The main disadvantage is performance: loading a fixed-length cryptographic key is usually more compact and faster than computing one at run time, which in our case may involve computing a hash value.

Although we believe that cryptographic hash functions and ciphers are more secure, we used a simpler XOR-based scheme – which satisfies our two properties – to minimize the performance cost in speed and size after embedding the crypto guards. We therefore do not claim that our encrypted code is cryptographically secure, but rather sufficiently masked to resist analysis and tampering attacks in a white-box environment, where the attacker has full privileges.

4.2 A Network of Crypto Guards

With *crypto guards* as building blocks we can construct a network of code dependencies that make it harder to analyze or modify code statically or dynamically.

A first requirement is protection against static analysis. Therefore, all functions in the binary image, except for the entry function, are encrypted with unique dynamic keys. We opted to decrypt the functions just before they are called and to re-encrypt them after they have returned. In this case, only functions on the call stack will be in the clear.

Secondly, as the key for decryption should be fixed, regardless of how the function was reached, we need to know in advance the state of the code from which the key is derived (encrypted or in the clear). Many functions have multiple potential callers. Therefore, we cannot always use the code of the caller to derive the key. The solution is to use a dominator in the call graph. As a dominator is by definition on the call stack when the function is called, it is guaranteed to be in the clear. We have chosen to use the immediate dominator to derive the key.

Note that it is also possible to derive the key of other functions, allowing one to create schemes which offer delayed failure upon malicious tampering [20]. On the one hand, this may allow a tampered application to run longer, on the other hand this does not correlate the moment of failure to the embedded checking or reaction mechanism.

Thus, a good mode of operation for the encryption (i.e. with error propagation) in combination with our dependency scheme, will propagate bit flips (triggered by tampering):

- through the modified function due to the mode of operation,
- inheritably from caller to callee according to the call graph due to the dependency scheme.

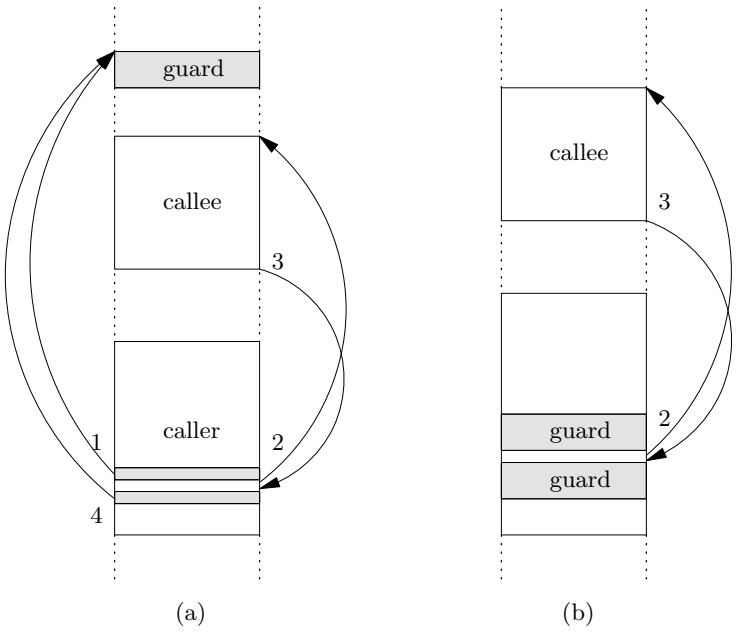

Fig. 1. (a) Memory layout of function call with calls to a crypto guard prior to the actual call and after its corresponding return. (b) After inlining the guard code. Note that the caller will increase in size depending on the size of the guard code.

The latter, however, does not validate for multiple callers due to our relaxation (using the dominator's code instead of caller's to derive the key) or to functions with no callees. In theory this could be solved by using authenticated encryption modes, such as EAX [2] or the more efficient OCB [16]. These modes aim to efficiently offer confidentiality and integrity.

The operation of a function call is illustrated in Fig. 1. It consists of the following steps:

1. the caller calls a guard to decrypt the callee;
   (a) the guard computes a checksum of the immediate dominator of the callee;
   (b) the callee is decrypted with the checksum as key;
   (c) the guard returns;
2. the caller calls the callee;
3. the callee returns;
4. the caller calls the guard to encrypt the callee;
   (a) the guard computes a checksum of the immediate dominator of the callee;
   (b) the callee is encrypted with the checksum as key;
   (c) the guard returns;

5 Numerical Evaluation

In our experiments we tested 5 benchmark programs out of the SPEC CPU2006 Test Suite [19] on an AMD Sempron 1200 MHz, running GNU/Linux with 1 GB of RAM. We first measure the impact of bulk encryption. Subsequently, we apply our on-demand encryption scheme where we protect a maximal number of functions, such that our scheme can offer tamper-resistance according to the properties mentioned in Section 4. To insert the guard code we used Diablo [7], a link-time binary rewriter, allowing us to patch binary code, insert extra encryption functionality, and perform dominator analysis on either the control flow graph or the function call graph. As we are generating self-modifying code, we mark all code segments to be readable and writable.

Our current implementation only handles functions which respect the `call-return` conventions. Recursive functions (denoted by cycles in the function call graph) are protected by decrypting ahead, i.e. just before entering a recursive cycle, one decrypts all functions part of that cycle.

To report the performance cost we define the *time cost* C_t for a program P and its protected version $\bar{P}$ as follows:

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

where $T(X)$ is the execution time of program X.

5.1 Bulk Decryption

For the bulk decryption we added a decryption routine that is executed prior to transferring control to the entry point of the program. For simplicity, we encrypted the entire code section of the binary (including library functions as Diablo works on statically compiled binaries). The resulting overhead in execution time is less than 1%.

5.2 On-Demand Decryption

On-demand decryption protects functions by decrypting them just before they are called and reencrypting them after they have returned. This limits their exposure in memory. Despite the simplicity of this scheme, a number of issues need to be addressed. An overview is given below.

Loops. A scheme considering only decryption should not be nested in a loop (unless it tests for the state – cleartext or ciphertext). Bulk decryption for example should happen only once. However, the sooner this decryption is performed, the longer code will be exposed. As our scheme operates on a function level and a corresponding function call graph, we do not posses information on loops, unless derived from further analysis (e.g., via profile information).

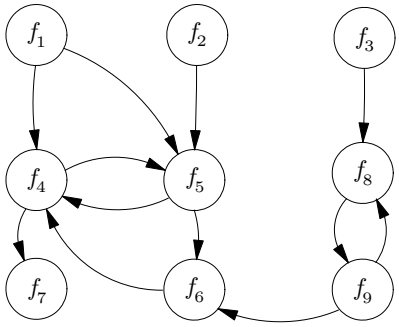

Fig. 2. A partial function call graph containing recursive cycles: $\{f_4, f_5\}$, $\{f_4, f_5, f_6\}$, and $\{f_8, f_9\}$. Functions f_1, f_2, and f_3 are the functions giving control to the recursive cycles after decrypting all functions in the reachable cycles.

Recursion. If a function calls itself (pure recursive call), it should – according to our scheme definitions – decrypt itself, although it might be in clear already. Therefore, we suggest decrypting 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. In this case, all functions in the recursive cycle should be decrypted before entering the recursive cycle. Figure 2 illustrates this. For example, before giving control to f_8 (via f_3), cycles $\{f_8, f_9\}$ and $\{f_4, f_5, f_6\}$ have to be decrypted. Function f_7 can be decrypted (before calling) in f_4 as it will always be re-encrypted (after returning) before f_4 calls it a second time (e.g., in another iteration of the recursion)

Multiple callers. In order to propagate errors through the whole call graph according to the call stack, decryption of a callee should depend on the integrity of all callers. This is not straightforward as we defined our scheme to rely on cleartext code, but only one of the callers has to be in clear. Cappaert et al. [3] is possible to decrypt the cleartext code of each caller. However, this requires $O(nd)$ decryptions (via a guard) where n represents the number of callers and d the difference in the call graph depth of each encrypted caller relative to the actual caller. To overcome this overhead we propose to apply a similar strategy as proposed for recursion, namely to decrypt ahead of the call to the callee. Thus, to decrypt a function b with multiple callers a_i we can decrypt the code of b before entering one its callers a_i. This would only result in $O(n)$ guards but expose f_b's code a little longer.

In our implementation we rely on the immediate dominator instead of the actual callers, but we only decrypt the code of b when it is called from one of its callers a_i. The reason is that the dominator is always on the call stack, and thus in the clear, when a function is reached. This only requires $O(n)$ guards.

Table 1 shows the time cost after applying our on-demand encryption scheme to 5 benchmark programs out of the SPEC CPU2006 suite. It is clear that,

Table 1. Time cost for on-demand encryption, using our tamper resistance scheme. This table shows the total number of functions, functions protected with the on-demand encryption scheme, the time cost, and the number of guard pairs (D and E) in the binary.

Program name	Functions total	Functions on-demand	Speed cost C_t	Number of guard pairs
mcf	22	20	1.09	28
milc	159	146	8.17	543
hmmer	234	184	3.20	873
lbm	19	12	1.00	20
sphinx_livepretend	210	192	6.65	1277

depending on the nature of the program (number of calls, function size, etc.), the impact of our scheme is moderate for some, while expensive for others.

5.3 Combined Scheme

To address the trade-off between performance and protection we propose a combined scheme. This scheme combines the merits of bulk encryption and the tamper-resistant properties of our on-demand decryption scheme. To decide whether a function is a good candidate for on-demand decryption, we define a heuristic *hotness* that is correlated to the frequency a function is called, namely:

Definition 1. *A function is* ***hot*** *when it is part of the set of most frequently called functions that together contribute to $K\%$ of all function calls.*

The call information was collected by analyzing dynamic profile information gathered by Diablo. This definition can be expressed by the following formula:

$$f \text{ is } hot \leftrightarrow calls(f) \geq threshold$$

with

$$threshold = calls(f_i) \mid \sum_{j=1}^{i} calls(f_i) > K \sum_{j=1}^{n} calls(f_i)$$

assuming n functions, ordered descending according to the number of times a function f_i is called, i.e. $calls(f_i)$.

When a function is hot, it is not selected for on demand encryption but protected by bulk encryption. Table 2 contains the time costs of the same benchmark programs tested when we apply the combined scheme. It is clear that defining a *hot* threshold reduces the overhead introduced by our guards. We believe that further fine-tuning of our threshold (e.g., increasing the K factor) will improve the performance of all programs.

Table 2. Time cost for our combined scheme, combining on-demand encryption with bulk encryption for $K = 0.90$.

Program name	Functions total	Functions on-demand	Speed cost C_t	Number of guard pairs
mcf	22	19	1.04	24
milc	159	135	1.95	486
hmmer	234	183	1.15	862
lbm	19	8	1.00	17
sphinx_livepretend	210	181	1.72	1257

Furthermore, we also would like to stress that we are aiming to protect all functions at all times, while most other software protection techniques focus on the critical parts only, or all functions but not at all times (e.g., bulk encryption).

6 Attacks and Improvements

6.1 White-box Attacks

Our guards, which modify code depending on other code, offer several advantages over the software guards proposed by Chang and Attalah [4] and the those from Horne et al. [9]:

Confidentiality. As long as code remains encrypted in memory it is protected against dynamic analysis attacks. With a good 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 at a later stage and thus will it most likely eventually crash due to illegal instructions. Furthermore, if the modification generates executable code, this change will be propagated to other functions, resulting in erroneous code.

Resistance to a hardware-assisted circumvention attack. This attack, proposed by van Oorschot *et al.* [22], exploits differences between data reads and instruction fetches to bypass self-checksumming 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. However, more recent work of Giffin et al. [8] illustrates that self-modifying code can detect such an attack and thus protect against it. As our work focusses on self-encrypting code, a type of self-modifying code, these results also apply to our techniques.

Nevertheless, in a white-box environment, an attacker has full privileges. For example, he or she can debug and emulate the program at will. This implies that our dynamically computed keys will be visible at some moment in time. The same counts for addresses of the decryption areas, etc. Therefore, we propose to protect guards in a diversified manner by obfuscation techniques [24] such that not all of them can be broken in an automated way. Another option is hardware support, such as cryptographic co-processors [21]. However, this usually comes a a higher cost.

6.2 Inlining Guard Code

Embedding a single decryption routine in a binary is not a secure stand-alone protection technique. It should always be combined with other techniques such as obfuscation or self-checking code. The strength of our scheme is a direct consequence of its distributed nature, i.e. a network of guards (as explained in Section 4.2). If implementation of the dependency scheme 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 preclude this attack and force an attacker to modify all instances of the guard code at run time, as all nested guard code will initially be encrypted. This has been illustrated in Figure 1(b). However, a disadvantage of this inlining is code expansion. Compact encryption routines might keep the spacial cost relatively low, but implementations of secure cryptographic functions are not always small.

Even though our initial results illustrated in Table 1 and Table 2 were performed by inlining calls, we expect similar performance results as our guard code in its most compact form does not exceed 40 bytes, while the calls we used for testing are 47 bytes long (pushing and popping arguments included).

6.3 Increasing Granularity and Scheme Extensions

Our scheme is built on top of static call graph information and therefore uses functions as building blocks. If one increases the granularity, and encrypts parts of functions, the guards can be integrated into the program's control flow which will further complicate analyzing the network of guards especially when 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, focusing instead on critical parts of the program and avoiding 'hot spots' such as frequently executed code.

As implied by Figure 1, the caller remains in cleartext as long as it is part of the call stack. Another extension involves encrypting the caller of a callee when the callee executes. This corresponds to protecting functions on the call stack. As such, only the executing function will be in cleartext. This extension

would double the number of guards per original function call inducing considerable overhead, see also [3]. However, using dedicated heuristics, such as *hotness*, would help us make a better trade-off between on-demand encryption and bulk encryption.

7 Conclusions

This paper presents a new type of software guards which are able to encipher code at run time, relying on other code as key information. This technique offers confidentiality of code, a property that previously proposed software guards 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 propose a scheme that makes code depending on one of its dominators. We compare our approach to the less secure bulk encryption. We introduce a heuristic based on the frequency that a particular function is called to reduce overhead. To validate our claims we implemented our scheme with Diablo, a binary rewriter, and applied it on 5 programs of the SPEC CPU2006 benchmarks suite.

Acknowledgements

This work was supported in part by the Research Foundation - Flanders (FWO Vlaanderen), the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen), the Concerted Research Action (GOA) Ambiorics 2005/11 of the Flemish Government, and by the BCRYPT Interuniversity Attraction Pole (IAP VI/26) programme of the Belgian government. We would also like to thank Elena Andreeva for her contributions.

References

1. D. Aucsmith. Tamper resistant software: an implementation. *Information Hiding*, 1174:317–333, 1996.
2. M. Bellare, P. Rogaway, and D. Wagner. The eax mode of operation: A two-pass authenticated-encryption scheme optimized for simplicity and efficiency. *Fast Software Encryption 2004*, 3017:389–407, 2004.
3. J. Cappaert, N. Kisserli, D. Schellekens, and B. Preneel. Self-encrypting code to protect against analysis and tampering. *1st Benelux Workshop on Information and System Security (WISSec 2006)*, 2006.
4. H. Chang and M. J. Atallah. Protecting software codes by guards. *Workshop on Digital Rights Managment (DRM 2001)*, 2320:160–175, 2001.
5. Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. Jakubowski. Oblivious hashing: a stealthy software integrity verification primitive. *Information Hiding*, 2578:400–414, 2002.
6. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report #148, Department of Computer Science, The University of Auckland, 1997.

7. B. De Sutter, L. Van Put, D. Chanet, B. De Bus, and K. De Bosschere. Link-time compaction and optimization of arm executables. *ACM Transactions on Embedded Computing Systems*, 6(1), 2007.
8. J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSA05)*, pages 23–32. IEEE Computer Society, 2005.
9. B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance. 2320:141–159, 2001.
10. M. Howard and D. C. LeBlanc. *Writing Secure Code, Second Edition.* Microsoft Press, 2002.
11. A. Klimov and A. Shamir. Cryptographic applications of T-functions, 2003.
12. 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.
13. D. Low. Java Control Flow Obfuscation. Master's thesis, University of Auckland, New Zealand, 1998.
14. N. Mehta and S. Clowes. *Shiva – ELF Executable Encryptor.* Secure Reality. `http://www.securereality.com.au/`.
15. A. Menez, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Inc., 1997.
16. P. Rogaway, M. Bellare, and J. Black. Ocb: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):365–403, 2003.
17. A. Shamir and N. van Someren. Playing "Hide and Seek" with Stored Keys. *Financial Cryptography '99*, 1648:118–124, 1999.
18. Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS07)*, pages 541–551. ACM, 2007.
19. SPEC – Standard Performance Evaluation Corporation. *SPEC CPU2006.* `http://www.spec.org/cpu2006/`.
20. G. Tan, Y. Chen, and M. H. Jakubowski. Delayed and controlled failures in tamper-resistant software. In *Information Hiding (IH07)*, volume 4437 of *Lecture Notes in Computer Science*, pages 216–231, 2007.
21. J. D. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. In *IP Workshop Proceedings*, 1994.
22. 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*, 2(2):82–92, 2005.
23. J. Viega and M. Messier. *Secure Programming Cookbook for C and C++.* O'Reilly Media, Inc., 2003.
24. G. Wroblewski. *General Method of Program Code Obfuscation.* PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.