Tamper Protection of Online Clients through Random Checksum Algorithms

Gisle Grimen, Christian Mönch, Roger Midtstraum
Department of Computer and Information Science
Norwegian University of Science and Technology
{grimen, moench, roger}@idi.ntnu.no

Abstract: We describe a new purely software-based, self-checking mechanism designed to prevent tampering of client programs in client/server-applications like online-games, peer-to-peer networks, or online auction systems. Our mechanism consists of randomly creating checksum algorithms at the server. The checksum algorithms are integrated into autonomous short-lived software code, called Mobile Guards, which are downloaded to a client program during its execution. The client programs are designed to be functionally dependent on the execution of a Mobile Guard. The randomly created checksum functions and the concept of Mobile Guards enable a highly dynamic protection mechanism, capable of rapidly evolving as new threats arise.

1 Introduction

A number of online services are structured as client/server-applications, where the client is executed on the user's computer. This gives the user the possibility to analyze and modify the client program. This can pose problems, for example in media streaming applications where data has to be kept confidential from the user, because a user might modify the client program in order to copy the media stream. Another example, where users might have an interest in modifying the client program are multi-user online applications like online games, peer-to-peer networks, or online auction systems. In online games user might modify the client program to gain unfair advantages over other users. In peer-to-peer networks users might modify the client to provide lower upload bandwidth and in auction systems they might be interested in automated bidding systems.

Such modification could result in a service becoming unpopular and as a consequence lead to great financial loss for the service providers. Service providers are therefore interested in protecting client programs against modifications. But in most cases they can, for example, not request the installation of special hardware. Users would not accept such an inconvenience in order to participate in an online game. Therefore those clients can usually only be protected by software-based mechanisms. The problem with software-based mechanisms is that they are executed in the same environment as the client program itself and are therefore as vulnerable to attacks, as the client program. A malicious user could analyze and modify the protection mechanisms, thereby circumventing it.

The attack on the protection mechanisms becomes possible in current solutions because they are usually based on a fixed set of mechanisms, for example one checksum algorithm. In addition, the protection mechanisms are usually embedded in the client program and are therefore fully exposed to an attacker. Such solutions can not solve the problem of tampering with clients. Especially not in a time, where one skilled person can create a patch to circumvent the mechanism and distribute this patch worldwide at virtually no cost.

In this paper, we present a new purely software-based solution to prevent user side modification of client programs. We enforce a continuous exchange of protection mechanisms between client and server. Each individual protection mechanism is only employed for a brief time interval and this time interval is too short to successfully analyze and attack the mechanism. Our solution can be instrumented to additionally detect modification attempts on client programs.

The remainder of the paper is structured as follows. In section 2 related work is discussed. The problem of client modifications in an untrusted environment and the goal of our approach are described in section 3. Our solution to the problem with the two main concepts of our approach, the mobile guard and random checksum algorithms are described in section 4. How random checksum algorithms are created and how mobile guards are protected is described in section 5 and section 6 respectively. Section 7 discusses how our mechanisms can be applied to counter dynamic attacks and section 8 concludes the paper.

2 Related work

Several mechanisms for creating tamper-resistant software have been developed. The majority of the mechanisms rely on some form of checksum algorithm to provide the software with a self-checking capability. We present a short survey over related work within software tamper-resistance below.

Aucsmith [Auc96] presents a method based on Integrity Verification Kernels (IKV). The IKV is embedded into the software and contains a set of tasks that are crucial for the program to function. Each task is divided into subtasks that are randomly grouped together in small segments called cells. Self-checking is achieved by including an accumulator function in each cell that gradually computes a result from the cells that have been executed.

Chang and Atallah [CA02] present a method of protecting software by embedding guards into a program. The guards can be programmed to do a certain tasks like calculating a checksum over the program code. The security comes from the multitude of guards used, resulting in a distributed network of independent guards that must all be neutralized in order to bypass the protection. A similar idea is present in [HMST01]. Their idea consists of embedding hundreds of linear hash functions in a program that each checks overlapping segments of the programs memory. They take great care in making sure their checksums are stealthy integrated into the program and that it is possible to embed a unique watermark into the program at install time without invalidating the self-checking mechanism.

Chen et al. [CVC$^+$03] present a different approach to checksumming software. Instead of calculating a checksum over the programs code, they calculate a checksum based on intermediate results from the program execution. The advantage of [CVC$^+$03] compared to [CA02] and [HMST01] is that the results from the executed instructions are checksummed and not just the sequence of instructions in memory.

The major problem with self-checking approaches like [Auc96, CA02, HMST01, CVC$^+$03] is that they are embedded into the programs they are designed to protect. Although one can apply several techniques that make them "stealthy" and obfuscate the program, it is impossible to prevent a determined attacker from circumventing the mechanism.

Kennell and Jamieson [KJ03] present a technique to establish the genuinity or trustworthiness of both the software and hardware of a computer system. They claim that by calculating a checksum over the instructions of the software along with side effects of the actual computation they are able to distinguish between a genuine and a modified system. The reason for this is that the side effects especially the memory hierarchy are difficult to simulate without large performance reductions. A later paper by Shanker et al. [SCT04] presents an attack on the Genuinity system presented in [KJ03]. Their attack relies on knowledge about the checksum algorithm. Without the time to analyze the system and to obtain knowledge about the checksum algorithm it would be considerable more difficult to successfully mount such an attack.

Seshadri et al. [SPvDK04] present a similar approach to Kennell and Jamieson for embedded devices. For their method to work they require knowledge about the hardware characteristics and the expected memory content of the device. They then verify the memory by traversing it based on a pseudorandom sequence. During the traversing of the memory a checksum is calculated and returned to an external part that verifies the computed checksum. The verification procedure is designed such that even the insertion of a single if-statement is visible to the external part. This approach is not well suited for von Neumann architectures because the checksum would have to include not only code but also all data, which is dependent on the current state of the device. The state of the device is usually not known to the external part, which makes it impossible to verify the checksum result.

3 Problem description and goal

For the scope of this paper we assume the following scenario. A service is created by a *service provider* and implemented on a *server* in a trusted environment. The service provider also creates a *client* program, that allows access to the service. *Users* of the service acquire and execute the client program on their computers. We assume, that the provided service requires a frequent, e. g. once a second, exchange of data between the client and the server. In addition we assume that the data that is exchanged between the client and the server does not have to be kept confidential from the user.

In order to guarantee that the service is provided as defined by the service provider, the service provider has to be considerably sure, that only unmodified client programs access the service. But since the client program is executed on the user's host and since the user has full control over the execution environment, he is able to modify the client program and change its semantics.

3.1 Attacks on client programs

If a user modifies the semantics of a client program we refer to this as an *attack* on the client program. We distinguish between two basic kinds of attacks on client programs, static attacks and dynamic attacks.

Static attacks A static attack is a modification of the client program that is completely performed, before the client is executed. A static attack is created with a static set of information, which has been gathered before the execution of the client. This information may include, for example, the client's program code, a trace of the protocol between the client and the server, or a runtime trace of the client.

Dynamic attacks A dynamic attack is a modification of the semantics of the client program that might use all information available in a static attack. In addition a dynamic attack uses additional information gathered during runtime of the client to modify the semantics of the client during its execution.

A typical static attack would be the application of a patch to a client program before it is executed. A number of such patches are available for online-game clients.

Is should be noted that dynamic attacks, in contrast to static attacks, usually demand considerably more computational resources on the user's side. A dynamic attack that monitors, for example, CPU register contents and modifies the execution semantics of the client accordingly, requires a sophisticated virtualisation environment. Depending on its capabilities and the underlying hardware, such an environment will slow down the execution considerably.

3.2 Goal

The goal of our research is to create *software-based mechanisms to protect client programs that handle non-confidential data against static attacks*. More specific the created protection mechanisms should have the following three properties in order to protect the client program:

Difficult to break It should be very resource intensive to break the protection mechanisms. That is, breaking should require much more resources than the average user is willing or able to spend.

Break once, break everywhere resistance It should not be possible to generalize an attack to the protection mechanisms. Every attempt to circumvent the solution should require the same amount of resources.

Facilitate detection The protection mechanisms should support the detection of attacks. This allows the service provider to react on attacks, for example by excluding the attacker from the service.

Typical examples for services that could be protected by our system are online-games, online-auctions, and peer-to-peer networks, because the success of these services depends on the provision of equal conditions for all users. We confine ourselves to the prevention of static attacks here, because we believe, that the average user of such a service does not have the necessary resources to perform a dynamic attack.

4 Mobile Guards and Random Checksum Algorithms

Our solution for protecting the client program against static attacks is based on continuously creating protection code on the server that is downloaded into the client. This code contains algorithms that are then executed in the client's environment and verify the client's integrity. We refer to the protection code as a *Mobile Guard*.

The task of a Mobile Guard is to verify the integrity of the client for a short time interval, which we refer to as a *trust interval*, before it is replaced by the next Mobile Guard. Each Mobile Guard contains a checksum algorithm that is randomly created by the server. The random creation of checksum algorithms ensures that they are unknown to an attacker and thereby prevent replay attacks. We refer to these checksum algorithms as *Random Checksum Algorithms* (RCSAs).

In order for a Mobile Guard to effectively prevent static attacks on a client, the following is necessary:

Protect the Mobile Guard against tampering The Mobile Guard has to be itself protected against analysis and modification by an attacker. It should not be possible for an attacker to extract the RCSA or to modify the Mobile Guard before it is replaced with a new Mobile Guard.

Ensure the execution of the Mobile Guard It has to be made sure, that the calculation of the correct checksum is necessary for the proper operation of the client. Together with the protection of the Mobile Guard, this will ensure that the Mobile Guard has to be executed.

In the remainder of this section we concentrate on how we ensure the execution of the Mobile Guard. The protection of the Mobile Guard is described in section 6.

Our approach ensures the execution of the Mobile Guard by making the calculation of the correct checksum a precondition for the proper operation of the client program. Since the user can not modify the Mobile Guard and since the random nature of RCSAs makes it impossible to predict the checksum, he has to perform the calculation to obtain the checksum. We therefore assume that the calculation of the correct checksum can only be performed by a Mobile Guard that is executed in an unmodified client program.

Given this assumption, there are a number of ways to make the knowledge of the correct checksum a precondition for the use of the client program. We elaborate two possible schemes in more detail, *checksum integration* and *checksum verification*.

Checksum integration In the checksum integration-scheme the server encrypts the data it delivers to the client program in such a way that it can only be decrypted with the correct checksum. The Mobile Guard decrypts all incoming data, using the calculated checksum as decryption key. As a result the data can only be accessed by the client, if the checksum is correct, i. e. if the client is unmodified.

Checksum verification The main idea of the checksum verification-scheme is that the server verifies the checksum. The Mobile Guard sends the calculated checksum back to the server and the server will only send further data, if the checksum is correct.

In the verification-scheme the checksum is removed from the context of the Mobile Guard and the client program and sent over an insecure communication channel. Due to this additional communication step we have to take additional measures to make sure, that the checksum was actually calculated by a Mobile Guard that is executed in the context of the client that is to be verified. If this is not ensured, a man-in-middle attack could be performed by copying the correct checksum from the communication of an unmodified client with the server and patching it into the communication between a modified client and the server.

To prevent this attack, we tie the checksum to the client in whose context it was calculated. In order to do so, the Mobile Guard hides the checksum in an opaque data structure. The Mobile Guard builds this structure by first choosing a random number that is then integrated into the checksum calculation. The resulting checksum and the chosen random number are then interleaved in a way that is specific to the Mobile Guard, yielding the opaque data structure. When the server receives this data structure, it extracts the checksum and the random number from it and verifies the checksum. If the checksum is wrong, the server stops sending data to the client. If the checksum is correct, the server encrypts the data sent to client so that it can only be decrypted with the random number. This ensures that only the client that created the checksum *and* the random number can process the data.

4.1 Integration versus verification

The main difference between the integration- and the verification-scheme is that the integration scheme does not require the client to send data back to the server. It can therefore be used in situations where users are concerned about their privacy and prefer the client to not “phone home”. The main disadvantage of the result integration-scheme is that the server can not detect modifications of client programs.

One advantage of the checksum verification-scheme is that it allows the server to detect modifications and to take appropriate measures, e. g. banish a malicious user. Another advantage is that the server can time the execution of the Mobile Guard in order to detect virtualisation attacks. Yet another advantage of the checksum verification-scheme over the checksum integration-scheme is that is does not use bandwidth to send data to compromised clients that will not be able to process the data anyway.

5 Random checksum algorithms

Randomly created checksum algorithms are a key component of our protection mechanism. In order to protect the client against tampering, RCSAs have to possess certain properties:

Deterministic RCSAs have to be deterministic to support their verification and their use as decryption keys.

Detecting RCSAs should have a fair probability to detect changes.

Diverse The result calculated by different RCSAs on the same input should be distinct with a high probability.

5.1 Detection probability with the application of n different checksum algorithms

If checksum algorithms are created randomly, their properties might not be completely known. This includes the probability for detecting changes. But even if a single checksum algorithm should not have a high probability for detecting a modification, the application of a number of different unknown checksum algorithms dramatically increases the probability for detecting changes. Let us assume that we created n RCSAs and that p_i is the probability that RCSA number i detects a change. Then the overall probability p to detect a change in a program if it is checked by all n RCSAs is: $p = 1 - \prod_{i=1}^{n}(1 - p_i)$. For every $p_i > 0$ the probability p grows towards one. Therefore we do not demand that every RCSA has a proven high probability to detect changes.

5.2 Creating Random Checksum Algorithms

There are many possibilities to create RCSAs. In our solution we concentrate on two of them, *input filter* and *function composition*. While it would be possible to create a random checksum algorithm by just appending a random number to the input of a known checksum algorithm, we did not use this approach. We favor the creation of different algorithms because it gives us more freedom in diversifying the Mobile Guards and therefore allows for a better protection of the Mobile Guard.

5.2.1 Creating RCSAs with input filters

The input filter approach is based on using an existing, well known checksum algorithm and modifying its input by passing the input through a randomly generated input filter. In order to create an RCSA with the input filter approach it suffices to randomly create input filters and combine them with the implementation of a known checksum algorithm. In our prototype we used input filters that divide the input data into blocks and permute the order in which theses blocks are provided to the checksum algorithm. As checksum algorithm we selected MD5.

To randomly create an input filter we create a random sequence of blocks and create a finite automata (FA) of type Mealy that outputs the order in which the blocks are to be read. The creation of the FA is itself randomized in order to increase the structural diversity of the implementation of the RCSA.

The creation of the FA starts from a given sequence b of blocks, an input alphabet Σ and an output alphabet Γ. It produces a FA and an input string w. If the FA is executed on the input string it will successively output the numbers of the sequence b. In our implementation the sequence numbers are encoded to base 10. Number representations are separated by the special character ϵ.

In order to increase the diversity of the created FAs, a number of their characteristics are randomly chosen. The digits for the sequence number representation in the output string are randomly selected from Γ. In addition the sequence number encodings, the output string contains “noise”, i. e. characters that are not used in the encoding of sequence numbers. If a transition is added to the FA, and at least one state without maximum fan-out exists, then an existing state will be reused with a probability of $\frac{1}{2}$. Otherwise, a new state will be created. At current the FAs are table-driven implementations with a fixed transition logic, only the tables are changed in the creation process.

The checksum algorithm performs transitions of the FA until the next number is completely output, ignoring all “noise” characters in the output. It then decodes the number and continues the checksum calculation on the appropriate block.

The process described here is just one way to create an input filtered checksum algorithm. An alternative way would be to decompose a known checksum algorithm and reassemble in a way that it reads the input in a given sequence. This approach would remove the static transition logic that our prototype uses.

5.2.2 Creating RCSAs by composing functions

This approach is based on randomly creating a checksum function from scratch. To create a checksum function, we split the input into parts of l words, a word is 32 bit wide. We then create a function f that reads l words from the input and m words from a variable area and outputs a word.

The function f is composed of *Elements*, *Terms* and *Expressions*. An *Element* is a word from the input block, a word from a variable area or a constant:

$$\text{Element} \quad := \quad \text{Input}[i] \mid \text{Variable}[j] \mid \text{Constant}$$

where $i \in \{1, \ldots, l\}$ and $j \in \{1, \ldots, m\}$ are randomly selected. Elements can be combined to Terms by applying a random number of the following transformations:

$$\text{Term} \quad := \quad \text{Element} \mid \text{Term} \wedge \text{Term} \mid \text{Term} \vee \text{Term} \mid \text{Term} \oplus \text{Term} \mid \neg\text{Term}$$

where $a \oplus b$ stands for: *a exclusive or b*. Terms are combined into Expressions by applying a random number of the following transformations:

$$\text{Expression} \quad := \quad \text{Term} \mid \text{Expression} + \text{Expression} \mid \text{Expression} \lll \text{Constant}$$

where $a \lll b$ stands for: *a circular shifted left b times*. During the creation of expressions, we allow only the operations *addition* and *circular left shift* to prevent reducing the dimension of the result. The result of every expression is assigned to a random variable:

$$\text{Assignment} \quad := \quad \text{Variable}[j] = \text{Expression}$$

The function f consists of a random number of assignments that are performed one after another. Note that the execution of assignment might change the variables. The result of the execution of f is the sum modulo $2^{32} - 1$ of all variables, i. e. $\sum_{j=1}^{m} \text{Variable}[j]$. Function f is applied to all parts of the input. The checksum is then the sum modulo $2^{32} - 1$ of all results of the application of f.

The composing functions-approach has the advantage that almost all code of the checksum algorithm is randomly created, leading to more structural diversity in the code. Since the building blocks of the code are quite small, it also allows for an easier interleaving with other algorithms than the input filter method. A disadvantage is that in contrast to the input filter-approach, the properties of the checksum algorithm are not known a-priori.

5.3 Properties of the checksum algorithms

We implemented generators for the input filter-based and the composing functions-based approach and tested their detection capabilities and their functional diversity.

Detection probability As the composing functions method randomly creates checksum algorithms, their ability to detect changes is unknown. To determine how well they detect modifications we generated 1 000 algorithms and 1 000 datasets each with only one bit changed compared to an original dataset of 1 MB. Each algorithm calculated a checksum on each of the 1 000 datasets, resulting in one million checksum results. The probability to detect a single bit change using our composing function-scheme was 0.99, which is more than sufficient, especially if considering that the algorithm will be replaced regularly. For completeness we also run the same test on the input filter method with the MD5 algorithm, which detected every change.

Functional diversity Another important property of our RCSAs is how functional diverse they are with respect to computing different results given the same input. For the input filter the diversity comes from the number of permutations of the order in which blocks are provided to the MD5 algorithm. If we have n blocks, we have $n!$ ways of providing them to the MD5 algorithm. The diversity of the composing functions comes from the random way in which data is processed.

To examine the diversity of our algorithms we generated $100\,000$ algorithms of each type and executed them on the same input of 1 MB. The number of blocks used in the input filter was 10. In our experiment, each of the $100\,000$ algorithms generated its own unique results, resulting in zero duplicated results. From our experiment we find that both methods provide more then adequate diversity, making the correct checksum sufficiently unpredictable to an attacker.

6 Protecting the Mobile Guard

As the protection mechanism is downloaded into the program, it is itself vulnerable to the same threats as the program. But any sensible attack to the Mobile Guard requires a thorough understanding of its algorithms and gaining this understanding requires knowledge, time and is usually a trial and error process [JL03]. To prevent intellectual attacks from succeeding, we create each Mobile Guards differently and restrict its lifetime. Because the algorithms in the Mobile Guards differ from one another, a new attack has to be performed on each Mobile Guard. By restricting the lifetime of a Mobile Guard to a few dozen seconds we are able to effectively prevent intellectual attacks on an active Mobile Guard.

The only feasible attacks on the Mobile Guard are then fully automated attacks. That automated attacks are possible at all seems surprising since we do not merely modify an existing program to create a Mobile Guard, but create completely new programs for every Mobile Guard. This freedom is the very reason, that a single intellectual attacks can not succeed. So why should an automated attack be possible? The answer is that the checksum algorithms are also created automatically and that the generators that create the checksum algorithms might introduce exploitable structures into the Mobile Guard. Such an exploitable structure could be a fixed memory location where the input filter stores the block sequence or a fixed memory location where the composed function stores the checksum result. We do not expect an automated attack to be able to autonomously identify exploitable structure. Instead an automated attack will rely on the intellectual identification of those structures. If we modify the structure of each Mobile Guard sufficiently, this kind of information will not be available and a fully automated attack will not be possible.

We vary the structure of the Mobile Guards by randomly positioning variables and by randomly positioning code blocks in the Mobile Guard. We are also considering to apply the techniques described in [CTL98, ASCB05, WHKD00].

Randomizing positions of variables An RCSA generator positions variables that are used by the Mobile Guard randomly in the Mobile Guard's memory. As a result, an attacker can not rely on the assumption, that a certain value is stored in a certain memory location. Instead the location of a variable has to be determined every time a new Mobile Guard is encountered.

Randomizing code locations An RCSA generator splits the Mobile Guard's instructions into basic blocks and positions them randomly in the Mobile Guard's memory. This includes the entry point of the Mobile Guard. In order to be able to transfer control to a new Mobile Guard, the RCSA generator creates the memory layout of the Mobile Guard number $n + 1$ when generating Mobile Guard number n. The entry point of Mobile Guard number $n + 1$ can then be encoded into Mobile Guard number n.

6.1 Structural diversity of the generated Mobile Guards

The input filter-based RCSAs were created with a static, table-driven implementation of the state transformations. The instructions of different Mobile Guards are therefore identical, except from the location dependent instructions, which are modified during the randomization of variable and code positions. The main differences between input filter-based RCSAs are due to the different transition tables. The tables of $1\,000$ randomly generated RCSAs with 19 input blocks contained between 76 and 142 different states and between 136 and 352 transitions. The input strings had the according length and all input strings differed from each other.

The composing function-based approach was examined by generating $1\,000$ composing function-based RCSAs. As expected, their code differed greatly from one another. The generated RCSAs contained between 597 and $3\,839$ instructions. Except from the first few bytes at the entry point, all RCSAs differed from each other.

Despite the random positioning of basic blocks in the different Mobile Guards, the entry point might still be found by applying a pattern based search on the instructions. This provides a starting point for the only attack that seems possible, the simulation of the execution of the RCSAs with the goal to calculate the checksum. But this approach would still require the automated identification of the memory location in which the checksum is stored and the identification of the moments in the execution, when the correct checksum value is stored in this location. We believe that it is unlikely that an automated tool can perform this analysis.

7 Remarks on dynamic attacks

While a pure static attack on the Mobile Guards is unlikely to succeed within the time limit, checksum algorithms are vulnerable to dynamic attacks because their operations are easily identifiable. Checksum algorithms perform data accesses to memory locations that contain instructions. These operations stand out from the all other operations performed by a typical program. Wurster et al. present a generic attack based on this observation [WvOS05, vOSW05]. The attack can be generalized by treating data reads and instruction fetches differently. Data is read from a memory region that contains an unmodified version of the program, while instructions are fetched from a memory region that contains a modified version of the program. The two protection mechanisms that are presented in [CA02] and [HMST01] are susceptible to this attack.

The self checking approach that is resistant to this attack is [CVC+03]. The reason is that this approach differs from typical checksum algorithms in the way that it does not verify the memory image of the instructions, but calculates a checksum over the side effects these instructions have on the data in memory.

A general approach to prevent the generic attack is to make the proper execution of the protected program depending on data accesses to the programs instructions, as described in [GMM05]. This approach makes it impossible for an attacker to use data reads on instructions as a criteria to identify checksum operations. Instead the attacker has to trace the calculations performed on the data that was read in order to identify its use, which is a very difficult problem.

8 Conclusion

We presented a new purely software-based, self-checking mechanism designed to prevent tampering of client programs in client/server-applications like online-games, peer-to-peer networks, or online auction systems. We take advantage of the fact that the clients have a continuous need to communicate with the server. By making the knowledge of the correct checksum a precondition for the access to the communicated data and by using random checksum algorithms we enforce the execution of Mobile Guards in the client program.

The Mobile Guards are protected against modifications in two ways. A short lifetime prevents intellectual attacks. Diversification of the memory layout and of the instructions of the Mobile Guard prevents automated attacks. Our generators have created Mobile Guards with significant differences in structure.

We presented two ways to create random checksum algorithms, the input filter-scheme and the compose function-scheme. Both methods generate checksum algorithms that have high probabilities to detect changes. In addition all generated checksums algorithms calculate different results on identical inputs, making them well suited for their application in Mobile Guards.

Our work shows that, although checksum algorithm alone can not prevent all threats to software, they are an important component. Their strength is that they are capable of verifying that the correct instructions are present and that they can be used to enforce the execution of additional protection mechanisms.

Further work will concentrate on additional methods to diversify the Mobile Guards in order to improve its resistance against automated attacks. This includes an investigation of the possibilities that the random creation of algorithms offers.

References

[ASCB05] Bertrand Anckaert, Bjorn De Sutter, Dominique Chanet, and Koen De Bosschere. Steganography for Executables and Code Transformation Signatures. In *the 7th International Conference on Information Security and Cryptology*, volume 3506 of *LNCS*, pages 425–439. Springer, 2005.

[Auc96] David Aucsmith. Tamper Resistant Software: An Implementation. In *Proceedings of the First International Workshop on Information Hiding*, pages 317–333. Springer-Verlag, 1996.

[CA02] Hoi Chang and Mikhail J. Atallah. Protecting Software Code by Guards. In *DRM '01: Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, pages 160–175. Springer-Verlag, 2002.

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Principles of Programming Languages 1998, POPL'98*, pages 184–196, 1998.

[CVC$^+$03] Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha, and Mariusz H. Jakubowski. Oblivious Hashing: A Stealthy Software Integrity Verification Primitive. In *IH '02: Revised Papers from the 5th International Workshop on Information Hiding*, pages 400–414. Springer-Verlag, 2003.

[GMM05] Gisle Grimen, Christian Mönch, and Roger Midtstraum. Software-based copy protection for temporal media during dissemination and playback. In *the 8th International Conference on Information Security and Cryptology*, volume 3935 of *LNCS*. Springer, December 2005.

[HMST01] Bill Horne, Lesley R. Matheson, Casey Sheehan, and Robert Endre Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance. In *Digital Rights Management Workshop*, pages 141–159. Springer-Verlag, 2001.

[JL03] Hongxia Jin and Jeffery Lotspiech. Proactive Software Tampering Detection. In *ISC*, pages 352–365, 2003.

[KJ03] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th USENIX Security Symposium*, pages 295–308, Aug 2003.

[SCT04] U. Shankar, M. Chew, and J. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[SPvDK04] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy, 2004.*, 2004.

[vOSW05] Paul C. van Oorschot, Anil Somayaji, and Glenn Wurster. Hardware-Assisted Circumvention of Self-Hashing Software Tamper Resistance. *IEEE Trans. Dependable Sec. Comput.*, 2(2):82–92, 2005.

[WHKD00] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report CS-2000-12, University of Virginia, 12 2000.

[WvOS05] Glenn Wurster, Paul C. van Oorschot, and Anil Somayaji. A Generic Attack on Checksumming-Based Software Tamper Resistance. In *IEEE Symposium on Security and Privacy*, pages 127–138, 2005.