Towards a Formal Model for Software Tamper Resistance

Cataldo Basile, Politecnico Di Torino
Stefano Di Carlo, Politecnico Di Torino
Thomas Herlea, Verizon Business
Jasvir Nagra, Google
Brecht Wyseur, Katholieke Universiteit Leuven

Abstract

A major challenge in software protection is the problem of tampering where an adversary modifies a program and uses it in a way that was not intended or desired. Several ad hoc techniques for software tamper resistance have been proposed, some of which provide a significant level of resilience against tampering. However, the literature lacks a formal definition of tampering that takes into account a model of the attacker's goals. One effect of this lack is the inability of easily comparing the actual efficacy of proposed tamper resistance mechanisms and evaluating the practical limits of tamper resistance.

This paper is a step towards addressing this shortcoming. We consider the two players: the defender who wishes to protect the program, and an adversary who wishes to modify the program as well as the assets embedded within the program to his advantage. We propose a way of expressing the intent of the defender and the attacker based on predicates defined over program traces. Based on these expressions, we present formal definitions for software tamper resistance, and software tamper verification. With a practical tamper verification scheme, we show how this formal model can be used in a reactive protection setting.

1 Introduction

Software tamper resistance techniques aim at increasing the ability of a program to continue its execution as intended even in the presence of an adversary who tries to monitor and change its behavior. Ideally software tamper resistance makes it intractable or costly for an adversary to modify sensitive components of a software.

Tamper-proofing is closely related to obfuscation where the intent is to make it difficult for the attacker to understand the program. In practice, the difference between tamper-proofing and obfuscation is labile since a program that is harder to understand due to obfuscation is often more difficult to modify. As with software obfuscation, the attack model for tamper resistance is one in which the adversary has direct access to the software implementation, and its execution platform.

The intuitive understanding of tamper resistance that includes preventing any tampering of the software is mistaken since an adversary can always alter or indeed completely replace a program. Nevertheless, common protection techniques such as verification techniques based on code checksumming [1] attempt to prevent any alterations of the program binary. A more nuanced definition of tampering must take in to account that there are some aspects of the program and its behavior (assets) that an attacker may want to preserve.

The execution of a program can be altered not only by modifying its implementation, but also by changing its execution environment. In the case of tampering, the execution environment is under the complete control of the attacker and thus any model for tamper resistance must contain both the environment and the implementation. For example, Van Oorschot *et* *al.* [2] demonstrated a tampering attack, where read, write, and execution of code could be distinguished from each other. As a result, read instructions could be altered such that a program's view of its own instructions could be different from the actual instructions allowing an adversary to modify the program with impunity.

The objective in this paper is to clarify the definition of tampering and tamper resistance. There are several players in this game – we have the entity who wishes to protect her program, the set of properties that she wishes to protect and an adversary who wishes to modify her program and has in his arsenal a set of analysis tools. Seminal work on formal definitions for tamper resistance have been presented by Canetti *et al.* [3] within the context of obfuscation, denoted as non-malleable obfuscation definitions. There too, a definition for tamper resistance, and a definition for tamper verification was presented. Our definitions agree with both respective non-malleable obfuscation definitions which enforces the soundness of our approach, but they distinguish in the fact that we present weaker definitions. As a result, where non-malleable obfuscation implies obfuscation, our definitions do not. Hence, impossibility results on obfuscation do not carry to our definition of software tamper resistance, which opens the door to possibility results for practical applications.

The approach we pursue is a way of expressing what the defending entity wishes to protect and what the attacker tries to achieve in terms of predicates on these execution traces. We describe an analysis of a proposed scheme which can be carried out by evaluating how effective the scheme is in protecting the assets of an application under a specific set of attacks. As a result of this clear separation between different assets and schemes which protect them, we hope that given an application with a different combination of assets, it will be possible to deduce the set of properties which can be protected with tamper resistance.

The remainder of the paper is organized as follows: In Section 2 we describe the behavior of the attacker, and it's capabilities. Within the scope of this attack model, which is common to any attacker that has control over the software we wish to protect, we formalize our concepts of software tampering in Section 3. We also present definitions on software tamper resistance techniques, and their relation to formal approaches in software obfuscation. Towards the end of our paper, a step towards run-time protection techniques is presented. Finally, we demonstrate an application of our formalization on a real protection scheme in Section 4. We conclude with a discussion of the strengths and short-comings of our proposed formalization in Section 5.

2 Attack Model

2.1 Some practical attacks

One of the biggest concern for a vendor of proprietary software (as opposed to Free Software / Open Source Software) is that unlicensed copies of the software will be made and distributed cheaper, depriving the vendor of license sales. An attempt to protect against such a phenomenon is made with software which, when executed, checks whether it is a correctly licensed instance and then decides whether to carry out its function or not. This check could be performed at startup or multiple times during the execution. An attacker could attempt to bypass this protection mechanism by modifying the code of the program such that the program's decision is always to carry out its function. The program modified in such a way is said to be a "cracked" version.

Sometimes vendors of proprietary software make available at no cost to the users a trial version of their software, which will carry out its function and will be equivalent to the full version for a limited time, after which it will refuse to run until the user purchases a license. The software may rely on environmental clues (system clock) in order to decide if it should stop functioning. Since access to these clues is mediated by the operating system and the operating system is under the control of the attacker, the attacker may configure the operating system to provide the program false clues, so the program always decides to continue functioning. In this case the attacker does not change the program, but tampers with the execution by manipulating the execution context.

In online games the problem of latency is often solved by keeping a part of the state locally at the client. In games where the player should not see the entire locally stored state because a part of it is pre-cached for potential future use, the attacker may try to change the program to reveal more of the local state than the game's rules allow. In strategy computer games "maphacking" reveals to the player portions of the map which have not been explored yet. In first-person shooters, "wallhacking" reveals to the player the scene which should be hidden by an opaque wall. These attacks are often achieved by adding attacker code to the graphics processing pipeline, for example by registering callback functions through the normal mechanisms provided by the graphics pipeline.

For various reasons a server may wish to rely on the client to abide by certain consistency principles:

- driving a vehicle in a game decreases its fuel,
- driving an in-game vehicle fast in a turn causes skidding,
- in a pay-per-use system micropayments are proportional with usage.

A client may be tempted to keep the program behavior they like and disable the one they do not like (consuming fuel, skidding, paying, ...). They might achieve this by running the program in an instrumented environment, such as a debugger, in which the beginning of the execution of an undesirable portion of the code could be intercepted and redirected to bypass it.

In developing the following attack model, we seek to be generic enough to support expressing any attack in a way that allows the feasibility of that attack to be tested and evaluated.

We caution, however, that a weakness of evaluating a scheme against a specific attack model is that it may be compromised by any attacker who manages to violate the assumptions it makes. With this in mind, we seek to make our model as generic as possible and clearly outline the assumptions we make.

2.2 Attack goals

As can be seen in the previous section the attacker could have, depending on the application, a variety of goals and a variety of techniques available for achieving them. It is therefore unfeasible to structure the analysis along the attacker's possible goals. We prefer a defender's perspective, similar to a risk-based analysis.

For the purposes of this paper, we only consider actions as an attack if their consequence is that the execution does not satisfy the business goals any more.

An important remark must be made about one particular case in which the attacker manages to produce an execution that does not satisfy the business goals at all: not executing the program. However, because we consider that to be every user's right, it is not considered an attack. More typically, an attacker will want to preserve some of the program's functionality, while disabling the rest.

2.3 Attacker capabilities

The attacker has no restriction on the tools and techniques to use to reverse-engineer and then to tamper with the application (e.g., super-user privileges are assumed to be available to the attacker). He/she can install any software on the target machine (e.g., debuggers, emulators). The attacker can read and write every memory location, processor registers and files. Network traffic and operating system are fully visible and changeable for the attacker. Moreover the attacker can start the malicious activity at any time, not just when the application is running.

Being in control of the target computer, the attackers can mount *environmental attacks* in which the program will be executed. System libraries and general purpose libraries are controlled by the attackers, along with the operating system. As a consequence, the attacker can use system calls, the input/output subsystem, the network stack, the memory management subsystem and possibly others for their purposes. Communication with any trusted entity is mediated by software that the attackers control, thus the communication can be read and changed by the attackers. The attackers can use virtualization, in which they control even hardware aspects visible to the program. An artificial environment can be presented to the software and an environment can be presented over and over again to the software, ignoring the passage of time and any change made by the software.

The attackers are also assumed to control the program storage medium for carrying out *static attacks* (attack is completed before execution). They can flip any bit of any file the program consists of, changing constants, the entry point to the program and machine code instructions. They can swap individual bits and groups of bits. They possess tools enabling them to do efficient comparisons and fast searches for bit patterns in the files (binary grep, search in a hex editor). Attackers possess tools for manipulating files at a higher conceptual level than strings of bits, tools that understand file formats and relationships between files. They are assumed to use tools that enable them to transform programs between different formats and between different levels of abstraction (disassemblers, decompilers). Concretely, with these means the attackers can change the entry point into the program, the values of constants used by the program, they can make the program jump over test instructions and change its functionality at will on the micro level.

By controlling the RAM and the CPU, the attackers are capable of performing *dynamic attacks*. At the finest level of granularity the attackers can use the CPU to investigate and intervene before and after the execution of any machine code instruction. This includes reading and changing the opcode and the operands. Moreover, they have the ability to let the program run unperturbed and interrupt it only when a certain condition is met. By choosing conditions which identify the moment in which an attack is easiest to carry out, the efficiency of mounting the attack is greatly increased. The attackers control the process memory (RAM, cache, processor registers, swap space), so they can read and change program constants, variables on the stack or on the heap. By controlling the program stack, the normal mechanism of function calls can be influenced. By reading the program counter the control flow can be mapped and by manipulating it, it can be altered.

The attacker can capture one run of the program and perform an identical replay. While not being interesting for attacking a text editor, as every time the same files would be created and edited, it is very interesting when the program's run resulted in playing a song or a movie. Any DRM policy seeking to limit the number of times a song or movie is played would be defeated by this execution replay attack.

2.4 Attacker limitations

The attacker has limited computing power available. The protection mechanisms used by the trusted entity are characterized by so called "security parameters" whose size, denoted k determines both the effort required to apply the protection technique and for breaking it. Good protection mechanisms enable the effort to apply the technique to grow only polynomially in

k, while causing the effort required to break the technique to grow exponentially in k. This captures the idea that with a relatively modest increase in the effort required to apply the protection, the attacker is forced to expend a disproportionally large effort for breaking it. How much more powerful the attacker is initially matters less than the fact that the trusted entity will be able to grow the effort required to break the protection faster than the attacker can keep up. For expressing this limitation we consider the attacker's power to be polynomially bounded by the size of the security parameter.

The security parameter could be derived from the length of the representation of the program and the length of the input consumed by the program during one run.

Since the attacker has access to the program, we will focus on how much this fact helps the attacker, as opposed to trying to perform an attack without such knowledge. Radically changing a program's execution requires more effort from the attacker than changing it a little if the results must still satisfy the attacker's goals.

Attackers are subject to the currently known laws of physics, especially to the limits derived from them. Concretely, attackers are assumed incapable of sending a message faster than the speed of light. Therefore, when the program interacts with a server using protocols that rely on timing, the server may be able to draw certain conclusions with certainty.

3 Formalization

A defender associates many business goals (assets) to software, each one to be preserved. What we would like to do here is to formalize these requirements, and to provide a unified view in a framework that can be used to compare and evaluate different software protection techniques.

One approach may simply be to test whether an adversary is able to modify any bit of the binary of a program executed on an untrusted platform. While this may initially appear to be the strictest possible requirement, it is not. In fact, the adversary has complete access to the machine, and he/she can modify the behavior of the application without altering the binary. For example, the attacker can use a debugger to dynamically modify some instructions or program state immediately prior to a portion of the program being executed. Therefore, to capture what is actually executed, we propose to use execution traces as a model of the behavior of a program on an untrusted platform.

An execution trace can be seen as a sequence of elementary computations, such as instructions that hit the CPU. It depends on the binary of the executable, its input (which may include input from remote entities), and the context of the execution (e.g., system libraries, operating system, architecture family). The values of the operands of these elementary computations are part of the trace, as well. When an attacker tampers with a program by skipping parts of it, or by executing them out of order, this will show in the trace with respect to the normal execution either as missing trace elements, or as reordered elements. If the attacker runs another program in parallel, such as a debugger or a virtual server, the trace will include the elementary computations belonging to that software, as well.

A set of execution traces associated with a program provides a basis for a formal theory on the semantics of programs. In computer science, this theory is often referred to as Abstract Interpretation [4]. We will use this as a basis for our approach to define software tampering. Let us denote

$$\mathcal{T} = \{\text{elementary computation}, \text{timestamp}\}^*$$

as the set of all possible traces of all possible programs. This is an infinite dimensional set which we will use as an abstract concept. Let us consider for example a program that performs an infinite amount of read operations. Each of these operations adds an additional dimension. It is possible to infer on traces to verify the execution of software. Our final goal is to verify the execution of programs running on untrusted platforms, and make the verdict whether or not the adversary has tampered with the application or its execution environment to the extend of his benefit. However, due to the nature of some classes of applications (such as networked applications), or the deployment of a particular protection technique (e.g., involving hardware as a secure execution environment), it is possible for the program to be split in different parts to be executed on different computing platforms. Therefore, we will need to produce a verdict based on the traces of all different parts. In order to link the different traces together, time references need to assist the elementary computations.

Although we are unable to capture the set $\mathcal{T}$ exactly, we are able to define predicates with $\mathcal{T}$ as domain. Let us denote by the "defender", the entity that wants to defend its assets over the targeted software, and D the predicate associated to his verdict. D is a predicate that expresses the defender's goals (protection of assets) by selecting the traces that satisfy those goals:

$$\begin{array}{rccl} D: & \mathcal{T} & \longrightarrow & \{0,1\} \\ & t & & \begin{cases} 1 \text{ if goals are met} \\ 0 \text{ else} \end{cases} \end{array} .$$

We assume that the program has no bugs, and in particular we assume the program free of security vulnerabilities. This implies that the program will never produce a trace that causes the D predicate to be false (within the intended context), regardless what inputs it is fed. The case of programs containing bugs, is much more complicated in the sense that several additional details need to be incorporated in the model, which are beyond the scope of this paper.

If the program is tampered with, but none of the functionalities that matter for the defender have been affected, the tampered program will produce traces that satisfy the predicate D. For example, a bank might not care about a modification that changed a program's background color, as long as all transactions and account information remain as they are intended to be.

On the other hand, we define the predicate $A : \mathcal{T} \to \{0,1\}$ to describe traces corresponding to successful attacks. The core statement here is that we only care about the attacker's actions as much as they affect the defender's goals. Therefore, we define 'useful tampering' as the modification of software or its execution environment in relation to the defender's predicate:

$$A = \overline{D} .$$

So far, D is a very generic tool. The upside is that it is capable of capturing a very wide range of business goals. The downside is that we can not make any statement about how feasible it is its construction starting from an arbitrary set of business goals, or how easy it is to compute the verdict, given the trace. In fact, the importance and the granularity of the (sub-)goals may even vary throughout the defender's organization. Other meaningful considerations can be done when we know the structure of the defender's organization. We will present as an example the case of D's whose structure is a conjunction of sub-goals. This however is extendable to different structures.

For programs that have a high level goal structure consisting of a conjunction of subgoals, D can be written as $D = (D_1 \wedge D_2 \wedge \cdots \wedge D_n) = \bigwedge_i D_i$, with $D_i : \mathcal{T} \to \{0,1\}$. This captures that all subgoals have to be satisfied in order for the aggregate goal to be satisfied. This notation however, fails to capture the idea that the users are allowed not to execute the program. Instead of "satisfy all subgoals", the aggregate goal should be "satisfy either all subgoals or none of them". The expression for D then becomes:

$$D = (D_1 \wedge D_2 \wedge \cdots \wedge D_n) \vee \overline{D_1 \vee D_2 \vee \cdots \vee D_n} = \bigwedge_i D_i \vee \overline{\bigvee_i D_i} ,$$

which leads to an attacker predicate of the following form:

$$\begin{array}{rcl} A & = & \overline{D} \\ & = & \overline{\bigwedge_i D_i \vee \overline{\bigvee_i D_i}} \\ & = & \bigvee_i D_i \wedge \bigvee_i \overline{D_i}\,, \end{array}$$

by De Morgan's laws. The factor $\bigvee_i \overline{D_i}$ corresponds to the intuition that if the attacker breaks any of the subgoals, the attack is considered successful. The factor $\bigvee_i D_i$ leads to the following paradox: "in order for the attack to be successful, at least one of the defender's subgoals must still be satisfied". As mentioned, from the defender's point of view, not executing the program at all and executing a completely different program are both allowed, they should not count as attacks. From the attacker's point of view, not executing the program at all or executing a version of the program altered beyond recognition will deprive him/her of any benefit from the original program, so again the attack should not be considered successful.

In fact, a consequence of the paradox is a limitation on the attacker's power. This idea also captures the worst case scenario for the attacker in software reverse engineering. The defender succeeds if the attacker cannot extract any valuable information from a protected software implementation, and is hence forced to start writing a program from scratch.

For example, when a home banking program is designed to do transactions, the defender (the bank) could have the following goals: debit the user's bank account (g_1); credit another bank account (g_2); and log the transaction (g_3). The bank wins if $D_1 \wedge D_2 \wedge D_3 = 1$, or when nothing (i.e., also nothing malicious) happened $D_1 \vee D_2 \vee D_3 = 0$. An adversary wins, if he for example was able to credit a bank account (preferable his), without debiting another (or at least not debiting his). Hence $A = D_2 \wedge \overline{D_1}$. This adversary does not care about the logging of the transaction. Although the operations in this example are atomically executed, this does not need to be the case in general. The essence is that the attacker and the defender share subgoals that they both want to be satisfied. In our example above, D_2 is a shared subgoal.

Now that we have an idea of how to capture the goals of the defender and the attacker based on traces, we need to describe the phenomenon of execution able to produce the traces. For this purpose, we introduce the concept of execution engine E as defined below:

$$\begin{array}{cccccccc} E: & \text{binary} & \times & \text{context} & \times & \text{inputs} & \longrightarrow & \mathcal{T} \\ & P, & & C, & & I & & t \end{array}$$

E encompasses the idea of "execution". It is not something concrete that can be attacked. E's output is the trace obtained by executing the binary P on the given context C with the presented input I. We can therefore assume that $D(E(P, C, I)) = 1$, where P is the binary program (provided by a trustworthy entity), C is the context-convention that was agreed upon (e.g., an Intel architecture with specific OS properties, system libraries, etc.), and I the provided input. This input may be a local input, or a remote input from some service. Note that, in order to be useful, the predicate $D \circ E$ needs to be computable, which is not the case in general (for example due to the halting problem [5]).

This concept enables us to formulate a definition for tampering directly to the program P instead of its execution trace. This corresponds to reality, where having access to the program's binary is of great advantage to the attacker.

Definition 1 (Tampering). *Let P be a program, with C the context on which the program should be executed. Then $D(E(P, C, I)) = 1$ for any legitimate input I.*

The program P is successfully tampered by an adversary, if in polynomial time complexity, the adversary can compute a $P' = P + \delta$ and/or find an execution context C', for which he can find an input I' such that $A(E(P', C', I')) = 1$.

C' captures the idea of executing a program in a "special" environment (e.g., virtual machine), while P' is the tampering of the binary itself. It is worth to note that, in our definition, we consider as tampering also those *modifications of the context preserving the program code.* What we say is that, in essence, tampering should be seen as a modification of the execution of the program, not solely modification of the program itself. I' captures the idea that an attacker finds "the magic input" (e.g., password that releases the programs protected section). Note that if part of the inputs are generated from a remote server, they are considered to be secure, due to the 'trust' we have for example in cryptographic primitives. However, an adversary can always deploy replay and various other protocol attacks, captured by I'.

To mitigate successful tampering attacks, adequate protection techniques need to be deployed. The tamper-protected program should have the same properties of the original one, based on the requirement that tamper-proofing does not alter the behavior. As a result, the adversary cannot produce a trace that fails the D predicate just by having access to the functionality of the program (this is often referred to as black-box access or oracle access). An attacker that can tamper and succeeds in producing a trace that makes D fail must have done it through tampering, since it could not have been done through black-box methods only. This brings us to a definition that constitutes tamper protection techniques.

DEFINITION 2 (Tamper protection technique). *A probabilistic Turing Machine $\mathcal{O}$ is a D-**tamper protection technique** for the class of programs $\mathcal{P}$ in the context C, if $\forall P \in \mathcal{P}$ all of the following properties are satisfied:*

- correctness – *$\mathcal{O}$ turns the program P into a functionally equivalent program $\mathcal{O}(P)$; there exists a polynomial p, such that if P halts in t steps, $\mathcal{O}(P)$ will halt in $p(t)$ steps, and $|\mathcal{O}(P)| \leq p(P)$.*
- soundness – *The probability that a polynomial-time adversary S given the code of $\mathcal{O}(P)$ is able to* tamper *with the program such that the defender's goals D are falsified in a way that a polynomial-time algorithm B given black-box access to the program P cannot do (running in the pre-fixed context C) is negligible. Formally,*

$$\left| \begin{array}{l} Pr[(P',C',I') \leftarrow S(\mathcal{O}(P),C,z) : A(E(P',C',I')) = 1] \\ -Pr[(P'',C'',I'') \leftarrow B^P(1^k,z) : D(E(P'',C'',I'') = 0] \end{array} \right| \leq neg(|P|,|I|)\,, \qquad (1)$$

the probabilities taken over the coin tosses of S and B; z denotes dependable auxiliary input.

The presence of auxiliary input in Definition 2 is required to capture any knowledge that the adversary has, prior to the attack. This is similar to the auxiliary input as defined in the context of definitions for obfuscation as by Goldwasser and Kalai [6].

To illustrate Definition 2, let us consider the following class of programs $\mathcal{P}$ that is related to Digital Rights Management. $P \in \mathcal{P}$ is a program, that accepts as input a password and an encrypted content. When the password is correct (D_1) the content will be decrypted (D_2), if not the program will terminate. The defender's goal is therefore captured by the predicate $D = (D_1 \wedge D_2) \vee \overline{D_1 \vee D_2}$. Assume P is compiled as follows:

$$\mathcal{O}(P)(x,c) = \begin{cases} \mathrm{Dec}_k(c) \text{ if } x = \alpha \\ \perp \text{ else,} \end{cases}$$

where α is the pre-set password that needs to be provided, and c is the encrypted content. Clearly, the compiler $\mathcal{O}$ is not a D-tamper protection technique since the adversary could overwrite the code of $\mathcal{O}(P)$ that verifies the password or modify the control flow at runtime (C'). As a result, the adversary can decrypt any given encrypted content with the tampered implementation without knowledge of the password. Hence, $A = D_2$. On the other hand, a black-box adversary has no chance to trick the program to decrypt without knowledge of α.

3.1 The Relationship between Obfuscation and Tamper Resistance

In the ideal case the function $\mathcal{O}$ would ensure all the necessary protection and the defender would not need to interact with the program during execution. One possible direction to achieve this, would be to obfuscate the program. If perfect obfuscation exists for a class of programs $\mathcal{P}$, it would have the potential to be $\mathcal{O}$. Formal definitions and results on obfuscation have been presented in the seminal work by Barak *et al.* [7]. Definition 3 captures the predicate-based definition.

Definition 3 (Predicate-based Obfuscation). *A probabilistic compiler $\mathcal{Q}$ is an obfuscator for the class of programs $\mathcal{P}$ if it satisfies the following three properties:*

- Functionality – $\forall$ *program $P \in \mathcal{P}$, $\mathcal{Q}(P)$ describes a program that computes the same function as P.*
- Polynomial size and slowdown – *There exists a polynomial p, such that $\forall P \in \mathcal{P} : |\mathcal{Q}(P)| \leq p(|P|)$, and if P halts in t steps on some input x, then $\mathcal{Q}(P)$ halts in $p(t)$ steps on input x.*
- Virtual black-box property – *For any predicate π and for any (polynomial time) adversary B, there exists a (polynomial time) simulator C, such that for all programs $P \in \mathcal{P}$:*

$$\Big| \Pr\left[B(1^s, \mathcal{Q}(P)) = \pi(P)\right] - \Pr\left[C_B^P(1^s) = \pi(P)\right] \Big| \leq neg(|P|)\,, \tag{2}$$

where the probabilities are taken over the coin tosses of B, C, and $\mathcal{Q}$, and s is some security parameter.

Informally, the *predicate-based* virtual black-box property states that, given access to the obfuscated program $\mathcal{Q}(P)$, an adversary should not be able to learn anything about the program itself that it could not learn from oracle access to P. This comparison between a 'real' world and an 'idealized' world was inspired by research in provable security, and is also used in subsequent definitions of obfuscation. We introduced this concept in Definition 2, since it enables us to compare the resistance of a protection technique in a 'tamperable' (real) world, and a 'non-tamperable' (idealized) world.

In [3], Canetti *et al.* introduced the concept of non-malleable obfuscation, which formally captures the concept of functional tamper resistance. Similar to our definition, Canetti *et al.* capture the effectiveness of an adversary's modification by a (polynomial-time computable) predicate relation. However, their definition of functional non-malleable obfuscation requires that a circuit (or program) cannot be tampered with in relation to any polynomial time computable relation. As a result, the definition of functional non-malleability is stronger than the obfuscation definition.

Intuitively, suppose $\mathcal{O}$ does not satisfy obfuscation for a program P, then there exists an adversary B that is able to compute a predicate $\pi(P)$ given access to the code $\mathcal{O}(P)$ which a black-box adversary (given oracle access to P) cannot compute. This is the unsatisfiability of the virtual black-box property (Equation 2). As a result, there will exist a relation that represents the effectiveness of the malleable adversary that can be achieved. Consequently, if we turn this around, an obfuscator that satisfies the functional non-malleability definition of

Canetti *et al.* [3], must also satisfy the predicate-based obfuscation definition of Barak *et al.* [7]. This result implies that any impossibility result related to predicate-based obfuscation carry over to the functional non-malleability case.

Our definition of tamper resistance is weaker, in the sense that it is defined in relation to specific attack goals. However, we claim that this is a realistic assumption to make, since we put this in relation to the defender's goals, i.e., the entity that wishes to protect the program from tampering. The consequence of the weaker definition is that we cannot prove any link with obfuscation any more. On the contrary, the opposite will be true, i.e., there exist reasonable classes of programs for which there exist tamper resistance techniques for reasonable defender's goals. Consider again a DRM example, augmented with a point function, implemented as follows:

$$\mathcal{O}(P)(x,c) = (y_1, y_2) = \left(\mathrm{Dec}_{\gamma \oplus x}(c); \begin{cases} s \text{ if } \mathcal{H}(x) = \beta \\ 0 \text{ else} \end{cases}\right),$$

where $\mathcal{H}$ is a pre-image resistant hash function, $\gamma = \alpha + k$ with k a decryption key for the decryption function Dec, and $\beta = \mathcal{H}(\alpha)$. Suppose that the defender's goal is to prevent the decryption routine to be executed when the authentication is incorrect. Then, $\mathcal{O}(P)$ is tamper resistant for this goal, since the adversary has only negligible possibility to guess γ correctly as k in order to use the decryption routine correctly. Under the predicate-based obfuscation definition however, $\mathcal{O}$ fails, since the secret s can be easily extracted from the source code, while an adversary having only oracle access would need to guess x correctly as α in order to do so, which is only possible with negligible probability.

On the other hand, it remains an open question, up to what extend obfuscation is satisfactory as a tamper resistance technique.

3.2 Reactive protection

While in the ideal case, the function $\mathcal{O}$ would ensure all the necessary protection, in practice a more feasible approach seems to be the implementation of a reactive protection mechanism. That is, a mechanisms which at run-time, verifies the execution of the application, and is able to react when tampering has been detected.

Such techniques however, need to compute a predicate (e.g., a tamper detection verdict) based on traces that are not yet complete. We will denote such a trace as a prefix-trace. Such a trace consists of t elements when prefix $= E(P, C, I)$, where the execution stops after t steps. The trace that contains the remainder of the execution is denoted as the suffix-trace. This brings us to the following definitions.

Definition 4 (Satisfactory Traces). *Let π be a predicate on traces. A finite trace prefix is called π-**satisfactory** if there exists a trace suffix such that $\pi(\mathit{prefix} \| \mathit{suffix}) = 1$.*

Definition 5 (Unsatisfactory Traces). *Let π be a predicate on traces. A finite trace prefix is called π-**unsatisfactory** if there exists a trace suffix such that $\pi(\mathit{prefix} \| \mathit{suffix}) = 0$.*

We are now able to express the requirements of a defender for run-time tamper detection in terms of properties of traces. For example, a defender that wants no change to the execution behavior of an application desires that the prefix-trace is D-satisfactory at any step in the execution. Intuitively, this means that at any point during the construction of a trace that can lead to satisfying the defender's predicate, it is possible to carry out an elementary computation that renders the attacker's goal unsatisfiable. In other words, it is possible to frustrate the attacker at every step during the execution.

In practice however, the definitions 4 and 5 are too strong to be meaningful. A defender might make a verdict at run-time that his goal is satisfied, given the prefix-trace, when he is convinced that the adversary has only negligible probability to find a suffix-trace that turns $\mathsf{prefix} \parallel \mathsf{suffix}$ into an trace that satisfies the adversaries goals: $A(\mathsf{prefix} \parallel \mathsf{suffix}) = 1$. This relates to Definition 2, where the task of the adversary is to tamper with the remainder of the execution of the program.

3.3 Evaluating effectiveness

One problem of defining the correctness of execution in terms of the execution trace *on an untrusted platform*, is that it is not immediately apparent how a verifying entity can learn from the trace. An omnipresent observer would be in a position to observe the operations on the client but the availability of such an observer would of course obviate the need for a protection mechanism in the first place.

There may exist local solutions that verify the execution of applications, but the verification techniques, and the subsequent tamper response are a potential target of the adversary. The cloning-attack by Van Oorschot *et al.* [2] is an example of how verification techniques can be defeated, while Tan *et al.* [8] suggested to delay tamper response as much as possible in order to hide the trigger to this response. Therefore, while there may exist solutions in the local case, we feel the defender will have a better chance of success with a solution where.

- the defender can monitor the attacker,
- the attacker depends on the continued cooperation of the defender in order for the program to achieve the shared sub-goals.

Let us denote by "the client", the platform where the application is executed and which is under full control of the adversary. "The server" is the platform where the defender verifies the behavior of the program and makes a verdict based on which he will continue to provide the service to the client (the server could be different physical entities). Schemes that enable such remote monitoring we will denote as *verification schemes.*

We have defined tampering in terms of what happens on the client since it is what we are trying to protect. *How* we are trying to protect the client is defined by each proposed verification scheme. Nevertheless in practice, every protection mechanisms will allow the client to provide *evidence* of the execution to the server. It is the task of the verification mechanism to introduce techniques that prevent the client from successfully forging this evidence undetectable.

We will call such evidence returned from the client to the server, a *tag*, and denote by TAGS the set of all possible tags. Let $G : \mathcal{T} \to \mathsf{TAGS}$ be a function which returns a tag for every trace prefix, and $V : \mathsf{TAGS} \to \{0,1\}$ be the predicate over the sequences of tags which tells the defender when the remote program can still be trusted.

$$\begin{tikzcd}
\mathcal{T} \arrow[r, "D"] \arrow[d, "G"] & \{0,1\} \\
\mathsf{TAGS} \arrow[r, "V"] & \{0,1\}
\end{tikzcd}$$

The effectiveness of a protection mechanism is how reliably a server can deduce from a given set of tags if the attacker has succeeded in tampering the application. That is, the following soundness property is needed to evaluate if $(V \circ G)$ is a good approximation of D:

$$\Pr\left[V(G(t)) \neq D(t)\right] \leq \mathit{neg}(|\mathsf{TAGS}|)\,,$$

where $|\mathsf{TAGS}|$ denotes the size of the tags (dimension of the image of G). Intuitively, this captures that the verifier has a higher level of confidence in the computed predicate when the tags are larger.

Let us discuss the cases where $(V \circ G) \neq D$: we distinguish the *false positives*, i.e., $(V \circ G)(t) = 0 \wedge D(t) = 1$, and *false negatives* $(V \circ G)(t) = 1 \wedge D(t) = 0$. False negatives are important from our perspective because they highlight the possibility of tampering with a program without being detected, not foreseen with an omnipresent observer for which $A = \overline{D}$ holds.

This brings us to the concept of a tamper verification technique, which is a protection technique that offers the adversary only negligible chance to *forge* a signature, i.e., to trigger a false negative.

Definition 6 (Tamper verification technique). *A probabilistic Turing Machine $\mathcal{O}$ is a D-* ***tamper verification technique*** *for the class of programs $\mathcal{P}$ in the context of C, if $\forall P \in \mathcal{P}$ the correctness definition of Definition 2 is satisfied, and the following soundness definition holds:*

(Soundness) – *The probability that a polynomial-time adversary S given the code of $\mathcal{O}(P)$ is able to defeat the tag verification (trigger a false positive at run-time) such that the defender's goals D are falsified in a way that a polynomial-time algorithm B given only black-box access to the program P cannot is negligible. Formally, $\forall i$ it needs to hold that*

$$\left| \begin{array}{l} Pr[(P', C', I', \gamma_i') \leftarrow S(\mathcal{O}(P), C, z) : A(E(P', C', I')) = 1 \wedge V(\gamma_i', z) = 1] \\ -Pr[(P'', C'', I'', \gamma_i'') \leftarrow B^P(1^k, z) : D(E(P'', C'', I'')) = 0 \wedge V(\gamma_i'', z) = 1] \end{array} \right| \leq neg(k)\,,$$

where the γ_i denote the i-th tag; the probabilities taken over the coin tosses of S and B; z denotes dependable auxiliary input; and k is a parameter in polynomial relation to $|P|, |I|$ and $|TAGS|$.

Note that the verifier expects γ_i to be $G(E(P, C, I))$ where E halts after i steps for some valid input I. The auxiliary information include in the verification can include for example knowledge obtained from previous tags that were verified.

In presence of an external observer, an attacker must cope with two tasks: trying to achieve his goal by tampering with the client program and hiding to the server that he is doing modifications (on the program, inputs or context). This is another strong limitation to the attacker's power. In fact, the attacker must concentrate on the class of tampering (P', C', I') that are undistinguishable under $(V \circ G)$ or develop an additional function γ to substitute the original evidence generator G. Non easy things when freshness is provided by inputs from the server. Furthermore, many attack techniques are based on "guesses" or rely on a trial and error approach [9], methods that are no longer possible when the attacker depends on the continued cooperation of a monitoring defender in order for the program to achieve the shared sub-goals.

We observe that our definition agrees with the verifiable non-malleable obfuscation definition by Canetti *et al.* [3]. The difference is that tags are included, the context is set towards run-time protection, and the attack predicate is narrowed down towards the defender's goals, which has a similar implication as noted in Section 3.1.

4 Invariants Monitoring

This section proposes an example of how the theoretical model presented in Section 3.3 can be used in a practical verification scheme for software applications based on the concept of *programming by contract* and *software invariants*.

Programming by Contract is an approach to designing software introduced by Bertrand Meyer in connection with his design of the Eiffel programming language [10]. Based on this model, software elements should be considered as implementations meant to satisfy well understood specifications or business goals (see Section 2) rather then simple sequences of executable code. The method provides formal, precise and verifiable interface specifications for software components based upon the theory of abstract data types and the conceptual metaphor of a business contract. The metaphor comes from business life, where a *supplier* (i.e., a software module, function, class, etc.) and a *client* (i.e., another software element requiring the service of the supplier) agree on a *contract*. The contract may

- impose a certain obligation to be guaranteed on entry by any client module that calls the supplier: the routine's *preconditions*. Preconditions are an obligation for the client, and a benefit for the supplier (the routine itself);

- guarantee a certain property on exit: the routine's *postconditions*. Postconditions are an obligation for the supplier, and obviously a benefit (the main benefit of calling the routine) for the client;

- maintain a certain property, assumed on entry and guaranteed on exit: an *invariant* property [11, 12].

Pre- and postconditions may be considered as special instances of invariants properties that should be guaranteed at well defined portions of the target module, i.e., prior to entry and at the exit. For this reason in the remaining of the section we will talk of invariants in general, including in this all three categories of conditions.

Invariants represent a way to describe the conditions on which a software element will work according to its specifications, and the condition it will achieve in return. As a consequence any violation of an invariant is a manifestation of a software implementation not respecting the original specification. Invariants are constructed as lists of boolean expressions, including the *true* assertion required to identify modules that do not require any specific contract.

Invariants play a central role in program development. Representative uses include refining a specification into a correct program, statically verifying properties such as type, declarations, and runtime checking of invariants encoded as assert statements [13, 14, 15]. Invariants play an equally critical role in software evolution. In particular, invariants can protect a programmer from making changes that inadvertently violate assumptions upon which the program's correct behavior depends.

4.1 Invariants Monitoring as a Tamper Verification Technique

The idea is to move from the detection of program modifications that *"inadvertently violate assumptions"* to the detection of program modifications that *"intentionally violate"* certain assumptions, i.e., program modifications that try to tamper with the correct behavior of the target application.

Lets us denote the *set of invariants* $I = \{I_1, I_2, \ldots, I_m\}$ as a set of predicates valid over a specific portion P^* of the program P to protect. These predicates are defined over a subset $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ of the whole set of variables computed in P^*. The "programming by contract" principle gives us an *a priori* information that these predicates must be true during every possible execution of P^*. This property can be used here to find a practical way to identify valid (D-satisfactory) traces. During the execution of P^*, each elementary computation may affect the value of variables in $\mathcal{V}$. If we select I in such a way that failing in respecting one of the defined invariants corresponds to failing in one of the defender's goals, we can actually exploit invariants to perform the defender task. If, at any point of the execution, some of the invariants I_i are not respected, we can therefore conclude that the program has been tampered with.

This actually means that the defender may continuously check the variable traces produced by P in order to detect any violation of the invariants defined in I. Since we assumed that the program is free of bugs (see Section 3), all traces computed over the target sets of variables need to satisfy I. If the program is tampered with, but none of the business goals were affected, the tampered program should still produce traces that satisfy the I.

It is clear that the ability of the defender to identify an attack is strictly connected to the considered set of invariants. The better is the considered set of invariants (i.e., the better he can identify violations of his goals) the higher will be the obtained protection. Besides considering invariants manually defined by the programmer, one can resort to automatically inferred invariants. Daikon provides an open source environment to accomplish this task [16]. Daikon is an implementation of dynamic detection of likely invariants. It discovers likely invariants from program executions by: (i) instrumenting the target program to trace certain variables, (ii) running the instrumented program, (iii) and inferring invariants over both the instrumented variables and derived variables not manifesting in the program.

Daikon implements a learning based approach to infer invariants strongly based on the set of inputs provided to the program during the learning phase. Given a real program, Daikon is able to infer thousand of different invariants properties defined through a formal grammar. Intuitively, such an elevated number of properties makes it difficult for an attacker to identify which subset of the properties is actually considered by the defender to construct his protection schema, and on the other hand, writing a tampered version of the program performing a different task w.r.t. the original one by preserving the full set of invariants defined by the application may result in a complex and time consuming task. Moreover, the set of invariants that one can automatically infer is strictly connected to the set of input used during the learning procedure. The invariants inferred by an attacker may differ from the one obtained by the defender, given the fact that the defender has an increased knowledge of the possible input domain. Pushing this concept to its extent, in a scenario in which the program requires a continuous interaction with a remote entity for its execution, it might be impossible for the attacker to infer all the invariants, as this process requires the program execution and therefore the interaction with the server.

Last considerations permit to assume that the probability of tampering with a program achieving the attacker goals and, at the same time, respecting all the invariants can be considered negligible (that is, soundness in Definition 6 is satisfied). Thus we can build a practical example of tags introduced in Section 3.3.

Let us define the trace predicate $J = J_1 \wedge J_2 \wedge \cdots \wedge J_m$, J_i being true if I_i is true after every elementary computation of the trace. The objective of invariant monitoring is to identify program's tampering by verifying the J predicate. If a trace t does not satisfy J then, any trace having t as a prefix won't be D-satisfactory (i.e., $D \Leftrightarrow J$ due to the correct choice of invariants). At any point of P^* execution, prefixes that do not satisfy J can be identified by inspecting the values that every variable $v_i \in \mathcal{V}$. For this reason, we assume as tags the value of variables in $\mathcal{V}$. We can therefore define G as a function that given a trace t provides the current value of the variables:

$$\begin{array}{rccc} G: & \mathcal{T} & \rightarrow & \mathsf{TAGS} \\ & t & \mapsto & \left(\left(\overline{v_1}, \overline{v_2}, \ldots, \overline{v_n}\right), \text{timestamp}\right) , \end{array}$$

where $\overline{v_i}$ is the value of the variable v_i at a precise time specified by *timestamp*. In the context of invariant monitoring, the verdict predicate V is defined as

$$\begin{array}{rccc} V: & \mathsf{TAGS} & \rightarrow & \{0,1\} \\ & \left(\left(\overline{v_1}, \overline{v_2}, \ldots, \overline{v_n}\right), \text{timestamp}\right) & \mapsto & I_1 \wedge I_2 \wedge \cdots \wedge I_m . \end{array}$$

That is, the verdict returns true if every invariant in P^* has been satisfied by the variable values

$(\overline{v_1}, \overline{v_2}, \ldots, \overline{v_n})$.

A practical implementation of invariants monitoring averts from the ideal case. In fact, tags are not generated and sent for verdict after every elementary computation. For this, it cannot notice of (very unlikely) tampering that violate invariants only between two tags solicitations (usually performed randomly so that the attacker cannot foresee and cope with them). Then, since invariants are properties defined over a specific portion P^* of P, in a real implementation the defender must be aware of which portion of the program is currently running in order to select the correct set of invariants and give a verdict. To make the formalization more clear and easily understandable, this "technical" information was not included. Nevertheless, it represents an advantage from the defender point of view. In fact, the attacker cannot simply tamper with the program by statically fixing the values of a certain number of variables.

5 Conclusion

This article presented a tentative of introducing a formal definition of the concept of software tampering, and tampering resistance. Our scheme proposes several innovative contributions to improve today's ad hoc tamper resistance techniques.

The intent of both the attacker and the defender has been modeled resorting to the concept of program traces. Execution traces are able to capture what a program actually executes by modeling its binary code as well as his state and context. Traces allowed us to formally define the concept of tampering in relation to the defender's goals, thus introducing the concept of useful tampering based on the defender's predicate. We have presented two definitions of tampering: one representing tamper protection techniques that capture the difficulty of the adversary to modify binary code in a sensible way; and a definition for tamper verification techniques which unable a verifier to react upon modification that conflicts with the defender's goals. These definitions provide a formal model that can be used to build practical protection techniques.

A first tentative approach of turning the proposed model into practice is presented, by introducing a tamper verification technique based on the use of program invariants monitoring. The work presented in this paper is a step towards addressing software tamper resistance. Future activities on this work focus on improved connections towards existing definitions of code obfuscation, and new software tamper prevention and verification techniques that can be proved secure under our proposed formal model.

References

[1] Bill Horne, Lesley R. Matheson, Casey Sheehan, and Robert Endre Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *DRM '01: Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, pages 141–159, London, UK, 2002. Springer-Verlag.

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

[3] Ran Canetti and Mayank Varia. Non-malleable obfuscation. In *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, volume 5444 of *Lecture Notes in Computer Science*, pages 73–90. Springer, 2009.

[4] David Schmidt. Lecture notes on abstract interpretation and static analysis.

[5] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[6] Shafi Goldwasser and Yael Tauman Kalai. On the Impossibility of Obfuscation with Auxiliary Input. In *Proceedings of the 46th Symposium on Foundations of Computer Science (FOCS 2005)*, IEEE Computer Society, pages 553–562, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (Im)possibility of Obfuscating Programs. In *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2001.

[8] Gang Tan, Yuqun Chen, and Mariusz H. Jakubowski. Delayed and controlled failures in tamper-resistant software. In *Information Hiding, 8th International Workshop, IH 2006, Alexandria, VA, USA, July 10-12, 2006. Revised Selcted Papers*, volume 4437 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2007.

[9] Nenad Dedic, Mariusz H. Jakubowski, and Ramarathnam Venkatesan. A graph game model for software tamper protection. In *Information Hiding, 9th International Workshop, IH 2007, Saint Malo, France, June 11-13, 2007, Revised Selected Papers*, volume 4567 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2008.

[10] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–585, October 1969.

[12] J.D. Fokker, H. Zantema, and S.D. Swierstra. *Iteratie en invariatie*. Programmeren en Correctheid. Academic Service, 1991.

[13] M.D. Ernst. Summary of dynamically discovering likely program invariants. In *Proceedings. IEEE International Conference on Software Maintenance, 2001.*, pages 540–544, 2001.

[14] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb 2001.

[15] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Proceedings of the 1999 International Conference on Software Engineering.*, pages 213–224, 1999.

[16] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.