A Taxonomy of Software Integrity Protection Techniques

Mohsen Ahmadvand[1], Alexander Pretschner[2], and Florian Kelbert[3]

[1,2] *Technical University of Munich*
[3] *Imperial College London*

Keywords— Tamper proofing, Integrity Protection, Taxonomy, Software Protection, Software Monetization

Abstract

Tampering with software by Man-At-The-End (MATE) attackers is an attack that can lead to security circumvention, privacy violation, reputation damage and revenue loss. In this model, adversaries are end users who have full control over software as well as its execution environment. This full control enables them to tamper with programs to their benefit and to the detriment of software vendors or other end-users. Software integrity protection research seeks for means to mitigate those attacks. Since the seminal work of Aucsmith, a great deal of research effort has been devoted to fight MATE attacks, and many protection schemes were designed by both academia and industry. Advances in trusted hardware, such as TPM and Intel SGX, have also enabled researchers to utilize such technologies for additional protection. Despite the introduction of various protection schemes, there is no comprehensive comparison study that points out advantages and disadvantages of different schemes. Constraints of different schemes and their applicability in various industrial settings have not been studied. More importantly, except for some partial classifications, to the best of our knowledge, there is no taxonomy of integrity protection techniques. These limitations have left practitioners in doubt about effectiveness and applicability of such schemes to their infrastructure. In this work, we propose a taxonomy that captures protection processes by encompassing system, defense and attack perspectives. Later, we carry out a survey and map reviewed papers on our taxonomy. Finally, we correlate different dimensions of the taxonomy and discuss observations along with research gaps in the field.

1 Introduction

A well-known attack in computer security is Man-in-the-Middle (MitM) in which adversaries intercept network messages and potentially tamper with their content. In this model attackers only have control over the communication channel and hence utilizing secure communication protocols can (to a great extend) mitigate the risk.

Beside network attacks, there is another type of attacks that could potentially lead to violation of safety and security of software systems. Given that end users have full control over the host on which programs are being executed, they can manifest serious attacks on software systems by inspecting a program's execution or tampering with the host's hardware/software [27]. These attacks are known as *Man-At-The-End* (MATE) attackers. As the name suggests, in this model the attacker resides at the end of line in which communication is secured and firewalls are bypassed. This is where the protection offered by security protocols for communication ends.

MATE attackers have direct access on the host on which the program is being executed. In most cases, there is no limitation whatsoever on what they can control or manipulate on the host. In effect, a user who installs an application on his or her machine has all controls over the computation unit. In this setting, perpetrators can inspect programs' execution flow and tamper with anything in program binaries or during runtime, which in turn, enable them to extract confidential data, subvert critical operations and tamper with the input and/or output data. Cheating in games, defeating license checks, stealing proprietary data (musics and movies), displaying extra ads in browsers are typical examples of MATE attacks.

Attacker goals and motives While there are different goals that MATE attackers may pursue, we particularly focus on attacks violating software integrity; other attacker goals are out of our scope.

Reputation, financial gains, sabotage and terrorism are non-exhaustive motivations for attackers to target the integrity of software systems. Disabling a license check is a classical example of MATE attacks in which perpetrator can potentially affect the revenue of the software vendor, for instance, by publicly releasing a patch for a popular software. Adversaries may harm the reputation of a company by manipulating program behavior, for instance, to show inappropriate ads. Perpetrators may further target the safety and/or security of critical systems. Corresponding consequences of attacks on a nuclear power plant system could be dire. Akhundzada et al. [3] elaborate on MATE attacker motives in more details.

Local attacks Local attackers normally have the full privilege on the software system as well as physical access to the hardware. Such attackers can readily tamper with softwares at any stage (at rest, in-memory and in-execution). Moreover, they could potentially load a malicious kernel driver to bypass security mechanisms employed by software. For instance, [89, 87] present two attacks to defeat protection schemes on a modified Linux kernel. Physical access to the host has the same effect. It enables attackers to tamper with systems' hardware and/or configurations. [35] shows that perpetrators can install a malicious hardware to disclose confidential data.

Remote attacks MATE attackers do not necessarily require physical access to the system of interest. Many harmful attacks can be carried out remotely. This type of attacks is known as *Remote-Man-At-The-End* (RMATE) [22]. In this model, attackers need to have either remote access to the target system or deploy their attacks by tricking end users. For example, Banescu et al. [8, 10] discuss two RMATE attacks on Google Chromium which are actually carried out by deployed malicious payloads into the Google Chromium browser. These payloads (in form of plugins) alter the original behavior of the browser to execute malicious activities, e.g. showing extra adds or collecting user information.

Intruders are another type of attackers that are very similar to RMATE attackers. Intruders penetrate software systems normally through their public interfaces (e.g. websites) in order to find vulnerabilities. A successful exploit may enable them to tamper with the integrity of the system, e.g. by materializing a buffer overflow or SQL injections. While RMATE attacks have much in common with intruders the main difference lies in the access privilege that RMATE attackers posses before and in order to manifest their attacks. Simply put, RMATE attacks exclude exploiting vulnerabilities and hence starts by a granted access to the host or a program that it contains. Nevertheless, an intruder (after successfully compromising the security) can carry out RMATE attacks. Therefore, the borderline between the two is blurry.

Further attack types Another realization of MATE attacks is through *repackaged software* [58]. In this attack, perpetrators obtain software bundles (normally popular ones) and modify

them with malicious codes to create counterfeit versions. Later, these repackaged softwares are shipped to software hubs to be installed by victims. Since malicious operations are normally dormant, counterfeit softwares appear to be the same as the original ones to end users. That is, they can remain on user devices for a period of time and eventually harm their assets, for instance, by deleting user files after 3 hours of program usage. Nevertheless, attackers first need to get users to install repackaged softwares. This, however, does not seem to be an obstacle for attackers: a recent study has shown that 77% of the popular applications available on Google play store, which is a trusted software repository for Android application, have repackaged versions [59].

Targeted malware is another form of MATE attacks in which a sophisticated malware is designed to violate the integrity of a particular system. Stuxnet [50] is a malware that manipulated programmable logic controllers' code causing severe damage to the Iranian nuclear program. ProjectSauron [51] is another example of targeted malware where governmental sensitive information were covertly collected and subsequently transmitted to the attacker's server.

In the light of these threats, researching, developing and deploying protection mechanisms against MATE attackers is of paramount importance. In particular, the integrity of software demands protection.

Approaches to protect software integrity Generally, integrity protection refers to mechanisms that protect the logic and/or data of particular software. Integrity protection is a part of the *Software Protection* field, which is also known as *tamperproofing*. Collberg defines tamperproofing as "[a way] to ensure that [a program] executes as intended, even in the presence of an adversary who tries to disrupt, monitor, or change the execution" [26].

At a high level, integrity protection mechanisms are comprised of two main components: `monitor` and `response`. While the former *detects* inconsistencies by monitoring a system's representation, the latter acts upon the detection of such inconsistencies by punishing the attacker [24]. In case of repackaged software and RMATE attacks, these reactions will be limited to terminating the process, informing users about violation of integrity or notifying a home server (phoning home).

Unlike cryptographic protocols, protection schemes against MATE fail to provide hard security guarantees, e.g., by difficulty of solving a computationally complex problem such as the discrete logarithm in a secure multiplicative group [73]. In fact, Dedic et al. [29] argue that there is no tamper resistant method that resists against polynomial time adversaries. This means all the protection schemes, given enough time and resources, are eventually defeated by attackers. However, protection schemes can raise the bar against perpetrators by increasing the cost and effort needed to violate system integrity. The introduced cost and effort is often sufficient to mitigate RMATE, repackaged software or targeted malwares. However, we are not aware of any studies that measures the resilience thoroughly.

Tamperproofing is commonly used in combination with *obfuscation* [27] software protection technique. Obfuscation aims to reduce the understandability of adversaries by complicating programs. In contrast to tamperproofing, obfuscation remains unaware of the program modifications. Moreover, tamperproofing can detect the occurrence of tampering attacks and respond to such attacks in some way punishing to actors [24].

Advances in hardware security has positively impacted integrity protection research. *Trusted Platform Module* (TPM) (`https://trustedcomputinggroup.org/`) is an approach in which software protection meets hardware security [68, 67]. TPM is a tamper-resilient micro-controller designed to securely carry out sensitive operations, such as secure boot, encryption, attestation, random number generation, etc. Recently, Intel has also been working on secure hardware

modules. Their Software Guard Extensions (SGX), introduced in 2015, are finding their way in academic research [11].

Gap Despite the existence of a multitude protection schemes that mitigate certain attacks on different system assets, we are not aware of a comprehensive study that compares advantages and disadvantages of these schemes. No holistic study was done to measure completeness and effectiveness of such schemes. How these schemes operate and what components they are comprised of not identified nor plotted in context.

Current classifications lack the level of detail required to evaluate and select schemes by practitioners. We are not aware of any classification beyond what was proposed by Collberg et al. [24, Chapter 7]. In effect, classifications were done only at an abstract level, i.e. monitor and response mechanisms. While integrity protection schemes comprise far more components and impact various parts of the system, they introduce unique constraints which may turn them completely inapplicable in certain application contexts. These limitations have left practitioners in doubt about the effectiveness and applicability of such schemes to their infrastructure.

Contribution We propose a taxonomy encompassing system, defense and attack views and extract relevant criteria in each view. These three views aim at capturing a holistic view of protection mechanisms. The system view captures the components of interest of the system to protect. The defense view elaborates on characteristics of defense mechanism. The attack view draws the attacker model and resilience of the schemes to certain attacks.

We evaluate our taxonomy by mapping over 49 reviewed research papers and discuss their advantages and disadvantages. We further correlate different dimensions of our taxonomy and discuss insightful observations with regard to security, resilience, performance, applicability and usage constraints in practice.

Structure This document is structured as follows. Section 2 presents a motivating data right management example to introduce further integrity threat samples, which supports the understandability of the taxonomy. In Section 3 we propose our taxonomy and define all its elements. In Section 4 we evaluate the taxonomy by applying it to the reviewed literature. To evaluate advantages and disadvantages of different schemes in Section 5 we correlate interesting dimensions of the taxonomy and discuss our findings. In Section 6 we review the related work. Section 7 discusses the future of integrity protection research. Finally, in Section 8 we conclude our study.

2 Motivating Example

In this section, we present a simplified digital rights management (DRM) system as a motivating example for our taxonomy. DRM enables authors to protect their digital proprietary data while sharing them with other people. To do so, DRM enforces a set of *usage policies* at the user end (client side) that limits what users can actually do with the protected content.

The goal of our sample DRM system is to enable secure lending of proprietary content to end-users. For the sake of simplicity, we assume that two usage policies are employed in this system: users can lease a protected media (digital content) **a)** for a certain period of time or **b)** for a particular number of views.

The system as depicted in Figure 1 is comprised of two compartments: a server (which runs on trusted commodity) and a client (which runs on untrusted commodity and hence is exposed to MATE attacks). On the server side, three components, viz. `license server`, `data provider (encryption)` and `proprietary data`, are collaborating to deliver the desired functionality. On

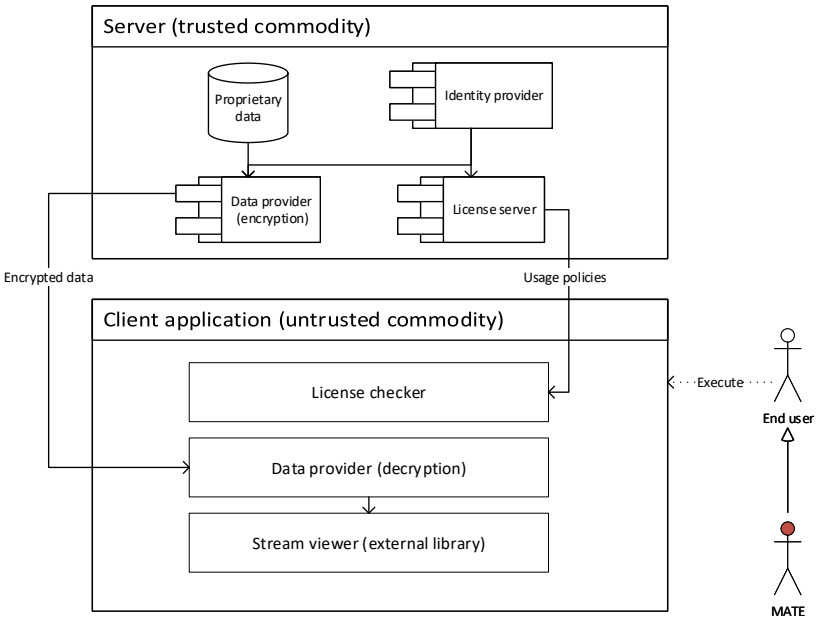


Figure 1: Architecture of a example DRM system to lend proprietary content.

the client side, three components, viz. `license checker`, `data provider (decryption)` and `stream viewer`, are handling user requests.

The `license server` is in charge of generating usage polices according to the purchased license by the users. It sends usage polices to the client applications. The `data provider` and `proprietary database` work closely together. The `data provider` retrieves the requested protected content from the database and encrypts it with client keys.

On the client side, the `license checker` enforces the policies that are specified by the `license server`. That is, the date of expiry and number of views are tracked by the license checker. As soon as the limit is met, the restriction is applied, for instance, by removing the protected content. To add more resilience, the data provider avoids to decrypt the file completely, instead it decrypts it in fragments and gradually feeds them to the stream viewer. In this way, the entire content is never exposed as a whole on the client side.

The entire system is developed by the *Sample DRM* company, except for the stream viewer. As stated in the figure, stream viewer is an external library, for which Sample DRM only has access to the compiled binaries.

Threats. In this system the protected content is transmitted to the client machines on which end-users have full control over program execution. This immediately gives rise to MATE attacks. In Figure 2 we present a set of potential threats from MATE attackers in form of an attack tree [55].

Starting from the root node, a perpetrator's goal is to use the protected content without the restriction which was specified in the usage policy. For this purpose, she has three options: extract data from the stream viewer, exfiltrate the content or circumvent the license checker.

Extracting protected content from the viewer requires the attacker to dump memory fragments and later reassemble them. She can (among other possible attacks) tamper with the viewer to perform this malicious activity automatically. Consequently, an attack can manipulate the viewer to dump and merge decrypted content behind the scene until the entire file is extracted.

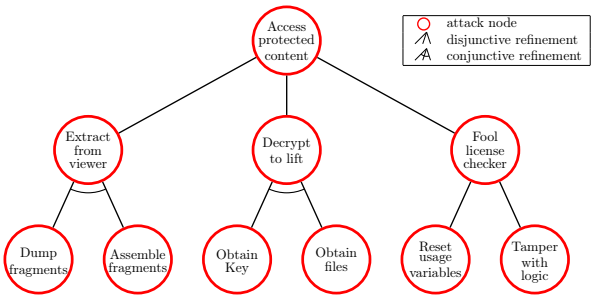


Figure 2: Attack tree capturing potential MATE attacks on the sample DRM system.

Exfilterating the content by decrypting requires the attacker to obtain the $client_{private}$ key as well as encrypted contents. Since the data provider (decryption) has access to this key, it can be targeted by reverse engineering attacks to first locate the key and subsequently to extract it.

Circumventing the license checker is yet another way to illegitimately use the protected content. The license checker is effectively in charge of verifying the usage policies on access requests. Attackers can manipulate the sensitive data, i.e. the usage variables for number of views and expiry date. If this is done, then the DRM protection is defeated. Furthermore, they can tamper with the license checking logic, for instance to accept any license as valid regardless of the actual validity of the license.

The aforementioned threats are a representative set of MATE threats that requires integrity protection in applications. Later in Section 4 we will map the elements of our taxonomy to the aforementioned example and show how our taxonomy serves as a blueprint for integrity protection practitioners. To avoid wordiness we refer to the aforementioned case study as the sample DRM in the remainder of this paper.

3 Proposed Taxonomy

Current classifications of integrity protection techniques lack a level of detail that is required by practitioners to evaluate and select schemes. To the best of our knowledge, the existing classifications remain at an abstract level, i.e. describing monitor and response mechanisms. However, integrity protection schemes comprise far more components and impact various parts of the system. For example, they may introduce unique constraints which make them inapplicable in certain application contexts. Such limitations have left practitioners in doubt about the effectiveness and applicability of integrity protection schemes to their infrastructure.

In this section we propose a taxonomy for software integrity protection techniques that provides a holistic classification and subsequently facilitates the usage of protection schemes in different contexts. We build the taxonomy on the basis of the protection process that a user follows when aiming to protect the integrity of a program. This process starts with identifying system assets and continues with the identification of possible attacks and defense mechanisms.

As a consequence, our taxonomy is comprised of three dimensions: **(i)** a *system view*, **(ii)** an *attack view* and **(iii)** a *defense view*. The *system view* describes the system as a whole, as well as the encapsulated assets, their granularity, and representations in different program life-cycles. It is the integrity of the assets that ought to be protected. Examples include license checkers and, in our DRM example scenario, data providers. The *attack view* captures actions that perpetrators may carry out to undermine the integrity of aforementioned assets. For example, Figure 2 shows that tampering with the logic of license checker could harm system assets. Finally, the *defense view* encompasses mechanisms to prevent or detect attacks on different representations of the

assets. For instance, we can utilize logic protection measures to mitigate the risk of tampering attacks on the license checker. Unlike attacks, defense mechanisms may have implications on the system life-cycle, since they may alter a system in various ways, e.g. in terms of performance, integration and incident handling.

In the course of building the taxonomy we noticed strong dependencies between the elements within and across the three dimensions. Because such relations are widely captured using class diagrams [45, 14], we make use of UML class diagrams [76] to model and depict such relations. Concretely, our model uses *association*, *inheritance*, *aggregation* and *composition* relations with the same meaning as defined in the UML specification. Association indicates two elements are related, simply put, one can access an attribute or a method of the other. Aggregation depicts the part-of relationship between two elements. Inheritance expresses specialization of a particular element, while composition expresses the set of elements that compose a particular element. In the following, we first introduce the taxonomy along with a few examples. Then, we discuss the dependencies between different views. Section 4 will later map related literature onto the classified attack and defense mechanisms.

3.1 System

The system dimension captures the characteristics of the system to be protected. This dimension is comprised of `asset`, `lifecycle activity`, `representation` and `granularity` main classes along with some more specific subclasses for each class. In the following, we first describe these main classes, their relation and then elaborate on each of them and their subclasses. Figure 3 presents the system view of the taxonomy.

`Asset` is the core class of this view as it captures the elements whose integrity need to be protected against attacks. In our taxonomy, assets include `behavior` and `data` the tampering of which may harm system users or software producers. Tampering with the license checking behavior and manipulating usage count variables data (number of views and expiry date) in the sample DRM are examples of behavior and data assets, respectively.

`Lifecycle activities` capture the different stages that a program undergoes in its lifecycle. Depending on the stage of a program, assets have different `representations`. Simply put, assets are exposed and presented in different representations from program source code all the way to the program code pages in the main memory. Each of these stages have access to different representation of the assets, however. That is, particular asset representations are only available in particular lifecycle activities.

An asset can have various `granularities`. Granularities correspond to different abstraction levels, e.g. a license check may correspond to a source code C function along with its set of statements, or, more fine-grained, a basic block within one function.

1. **Integrity assets** include valuable *data* or sensitive *behavior(s)* of the system, tampering with which renders the system's security defeated.
   (a) **Data** refers to any sensitive information that is produced or processed by the application. Application input, configuration and database files are examples of such data assets. For instance, in the sample DRM, usage count variables are data assets.
   (b) **Behavior** is the effect of program execution. Similar to tampering with data, subverting an application's behavior (logic) can have obnoxious consequences. For instance, in the sample DRM, stream viewing decrypted fragments as well as license checking are behavioral assets.
2. **Representation.** Assets can have multiple representations, viz. *static*, *in-memory* and *in-execution*, depending on the program state from start to finish.

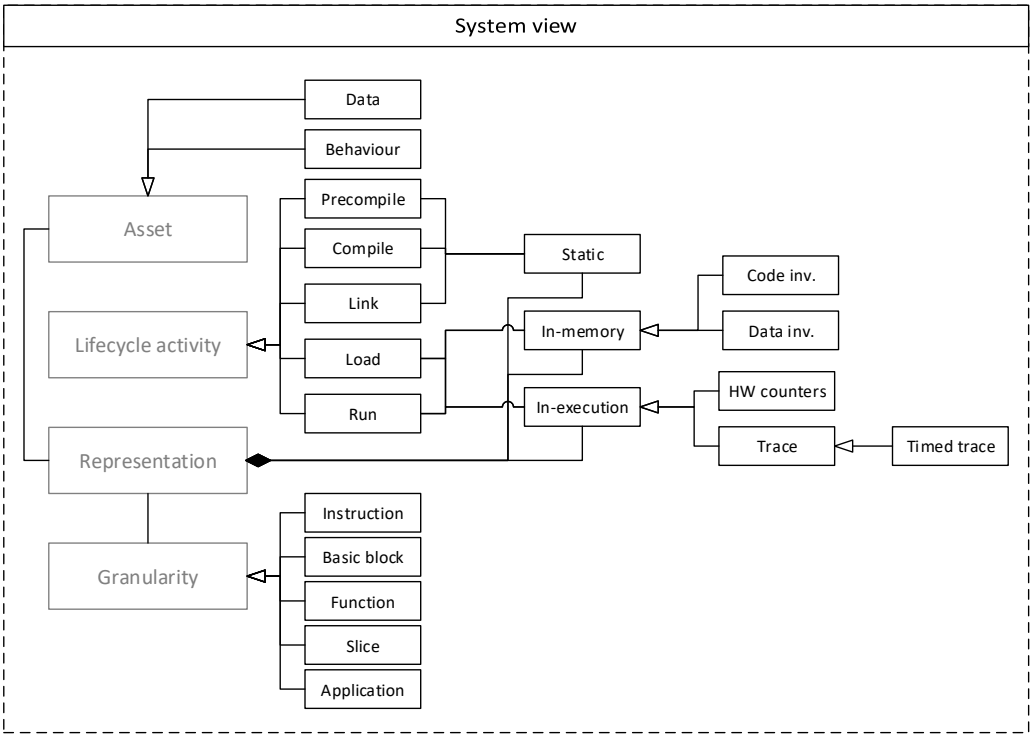


Figure 3: System view dimension of the taxonomy depicting the system main classes as well as subclasses and their relations.

(a) **Static** captures assets when they are at rest, i.e. stored on disk. These assets representations are accessible via the file system.

(b) **In-memory** represents assets in memory, i.e. RAM or virtual memory. Process memory captures this representation which includes *code* and *data* invariants of processes.

(c) **In-execution** captures tangible effects of assets during execution. *Trace* and *hardware counters* are subclasses of the in-execution representation. Meanwhile, trace has a subclass that comes with timing data, i.e. *timed trace*.

3. **Granularity.** Assets have different scales and levels of detail that vary from an instruction to the entire application. For example, a license check is usually represented as a function whereas control flow integrity depends on a large set of branching instructions in the application.

(a) **Instruction.** A single instruction is evaluated as security critical.

(b) **Basic block (BB).** A set of consecutive instructions with exactly one entry point and one exit point.

(c) **Slice.** A set of consequential instructions scattered in a program, e.g. the branching instructions that dictate the control flow of a program.

(d) **Function.** A complete function, e.g. *licenceCheck()*, is the goal of protection.

(e) **Application.** The entire application or library is supposed to be protected, i.e. the entire program is security critical. For instance, a power plants controller application is in its entirety sensitive.

4. **Lifecycle.** The lifecycle indicates a series of different states that each program undergoes from development all the way to execution, viz. *pre-compile*, *compile*, *post-compile*, *link*, *load* and *run*. Every program has to go through these activities strictly in the mentioned order in order to eventually get executed. However, if a program is not developed in-house, i.e. source codes are not at hand, the first two activities (pre-compile and compile) are out of reach on the protector side, as these activities require access to source codes.

   As depicted in the system view (Figure 3) by association links, the first four states (pre-compile, compile, post-compile and link) deal with the static representation of the assets. The last two states (load and run) are concerned with both the in-memory and in-execution representations. In the following we describe these states:

   (a) **Pre-compile.** This is the state in which a program's artifact, which are represented statically, e.g. in the form of application source code and other bindings such as testing, database and setup scripts, etc, are delivered. Source-level transformations could be triggered at this stage to add protection routines.

   (b) **Compile.** In this stage, a compiler transforms the program's (static) source code into the targeted machine code. It runs several passes starting from lexical analysis and finishing with binary generation. Additional protection passes can be integrated into the compiler pipeline.

   (c) **Post-compile.** After compilation, static program artifacts are transformed into executable binaries or libraries. Since the source code is no longer available in this phase, protection schemes operating at this level have to utilize disassembler tools to recover a representation (normally in assembly language) of the application on which they could carry out protection transformations.

   (d) **Link.** Refers to the state in which a *linker* obtains a set of static compiler-generated artifacts, combines them into a single binary and relocates address layout accordingly. Protections in this phase not only could carry out transformation on the program and all its (static) dependencies, but can also mediate (and potentially secure) the process of binary combination and address space management.

   (e) **Load.** This is the stage in which the *loader* (i.e. provided by OS) loads a previously combined executable into the memory, resolves dynamic dependencies and finally carries out the required address relocations. Unlike the previous states, this state deals with both the in-memory and in-execution representations of assets. More importantly, it is visited every time a program is subject to execution. Therefore, load time protection transformations can turn protection into a running target to render attacker's previous knowledge about the protection irrelevant. Loader transformations, apart from being executed on every program start, can also verify (and potentially protect) dynamically linked dependencies.

   (f) **Run.** This stage begins as soon as the loader triggers the program's *entry point* instructions and lasts until the program is terminated. Run also operates at the in-memory and in-execution representations of the assets. Moreover, the run state can constantly mutate a program to alter off-board or on-board protections. Although at

[0]Except for dynamic dependencies.

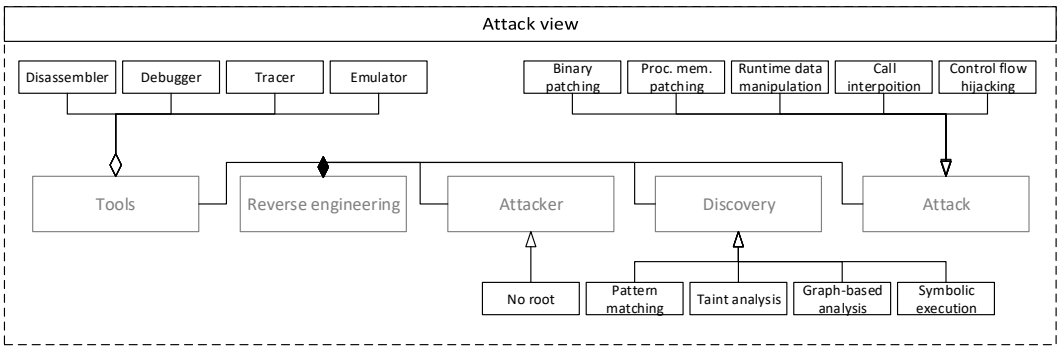


Figure 4: Attack view dimension of the taxonomy showing the attack main classes, subclasses and relations.

runtime dynamic dependencies are already resolved, still a protection mechanism can authenticate the loaded dependencies and decide upon unloading or reloading them.

The system view captures critical elements along with their various representation in the system that attacker have interest in undermining their integrity. Both the attack and defense views are applied on a system and hence the system view is the base of the taxonomy. In the following, we first elaborate on the attack view aiming for identifying potential threats and techniques to violate system integrity assets. Later on we discuss the defense view that provides means to mitigate or raise the bar against identified threats (from the attack view) and common attack tools.

3.2 Attack

The attack dimension expands over the *attacker* view, encompassing high-level attacks, tools and their relation to other dimensions of the taxonomy, viz. system view and defense view. As depicted in Figure 4, the attack view is comprised of the `Attacker`, `Reverse engineering`, `Tools`, `Discovery` and `Attack` main classes. Attacker represent the perpetrator whose goal is to harm system assets by violating system integrity. To do so, attackers use reverse engineering techniques. Reverse engineering is a process in which an attacker utilizes (offensive) tools to discover assets and possibly protection mechanisms. The process normally ends with manifestation of a concrete attack that harms system integrity after defeating protection measures (if any).

In the following, we discuss each main class of the attack view in more details along with their subclasses.

1. **Reverse engineering (RE).** RE is located at the heart of the attack view. It encompasses attack, attacker, discovery and tools. RE aims at harming system assets.

2. **Attack.** Tampering attacks themselves can have multiple forms (inheritance) based on which representation of the assets they are applied to. Attacks are carried out on a representation of the assets, thus there is a dependency between attacks and system representations. In the following, we elaborate on different attacks on different asset representations.

   (a) **Binary patching.** A successful exploit at the file system level can tamper with the static representation of the assets.

(b) **Process memory patching.** An exploit at the program level or OS level may enable attackers to directly tamper with the process memory.

(c) **Runtime data manipulation.** Similar to memory patching, program or OS level compromises can enable an attacker to tamper with the program's dynamic data that includes the input to a program, the produced output by a program. The runtime data is allocated in system stack and heap.

(d) **Call interposition.** Once a system is compromised, attackers may, for instance, intercept system calls to inject malicious behavior. Call interpositions are out of the scope of software protection, they rather fall under infrastructure security. For this reason, we did not include them in our paper survey. But we still believe it is a potential threat to software integrity and hence should be listed in the taxonomy.

(e) **Control flow hijacking.** Calls and branches define the execution flow of the program. Attackers target the program control flow in a wide range of attacks (e.g. return-oriented programming [80] and buffer overflows).

3. **Discovery.** In order for attackers to violate the integrity of a protected system they need to identify assets or protection routines in a given representation of a program. We name this phase the *discovery* phase. This implies a relation between the system representation and discovery.

   Banescu et. al [9] formulated attacks as search problems (e.g. identifying protection guards in the application can be formulated as a search problem). To the best of our knowledge, no study has addressed the difficulty of discovering protection techniques. In our survey, however, we found four techniques that are commonly referred in the literature, viz. `pattern matching`, `taint analysis`, `graph based analysis` and finally `symbolic execution`. Therefore, we resort to these four common approaches in discovering protection measures.

   (a) **Pattern matching.** Manually analyzing a large and complex program is labor and resource intensive. Therefore, normally attackers try to identify and defeat protection mechanism by employing automated attacks based on pattern matching, using for example *grep*. Pattern matching is not limited to search for strings, it also can search for properties of an artifact (e.g., entropy). Such attacks are plausible if and when an application commits to the usage of a recognizable pattern in their protection. Note that pattern matching can be applied on all representations.

   (b) **Taint analysis.** Tainting analysis is a technique in which the influence of a particular input on program instructions can be examined. This tool can facilitate the detection of protection routines for perpetrators. For instance, taint analysis can help attackers to find the connection between check and response functions by following the influence of check variables on response instructions [75].

   (c) **Graph based analysis.** A protected program can be represented as a graph in which basic blocks are the graph nodes and jumps express edges. Dedic et al. in [29] argue that in most cases protection nodes are weakly connected as opposed to the other nodes and hence easier to detect.

   (d) **Symbolic execution.** Enables adversaries to see which inputs to the program triggers the execution of which part of the program. Since the program is actually being executed to discover execution paths, nothing can remain unseen for symbolic execution engines as long as the constrain solver is successful. This unique feature of symbolic execution can enable attackers to visit all hidden (obfuscated/encrypted/dynamically loaded or generated) instructions of the protection scheme.

4. **Tools.** An attacker can utilize a set of tools to support reverse engineering activity, e.g. to carry out attacks or to identify assets or defense measures in place. To the best of our knowledge, the resilience of different schemes against reverse engineering tools has not been studied. Therefore, in our taxonomy we made a set of assumptions to decide whether to mark a scheme resilient against a particular tool or not. We will discuss these assumptions with substantial details in Section 5.7. In the following we discuss some generic tools that reverse engineers normally use.

   (a) **Disassembler.** An attacker may utilizes dissembler to disassemble a binary potentially to analyze the protection logic.

   (b) **Debugger.** Another tool that enables the attacker to monitor the execution of a program in a slow paced sequential manner. In this attack, debugger can access or alter any runtime data.

   (c) **Tracer.** Analyzing program's execution traces could potentially reveal protection mechanism. This becomes more useful when a program employs obfuscation/encryption to hide its logic. However, the traces can reveal the executed instruction (after decryption and de-obfuscation). In this event, attackers may employ more intelligent analysis to defeat a stealthy protection.

   (d) **Emulator.** Attackers can employ emulators to study program execution in a lab manner. All system calls and executed instruction can be closely monitored and thus deepen the knowledge of the program internals. With the help of snapshots, steps could be reversed to recover from faulty states, which in turn facilitates the attack.

5. **Attacker.** The actor who carries out disrupting actions to violate the integrity of the system is the attacker. When classifying attackers, distinguishing RMATE and MATE attackers, although appealing, is out of reach. The reason being that the two have very much in common, the main difference is the physical access that MATE attackers possess. Therefore, we classify our attackers to two groups: *attackers with root privileges* and *attackers without root privileges*. This is indicated with `no root` title (corresponding to attackers without root accesses) in our taxonomy in Table 2.

The attack view captures potential threats along with the tools and techniques that attackers can use to violate system integrity. The same attacks could be used by adversaries to circumvent or even defeat protection mechanisms as well. Therefore, in order for defense mechanisms to be effective, it is crucial that they resist against attacks and tools.

3.3 Defense

This dimension can be seen as the bridge between the system and attack dimensions. It serves as the core of the taxonomy by capturing the activities in which system assets are protected using a set of measures against potential threats. As can be seen in Figure 5, the defense dimension is comprised of four main classes: `measure`, `protection level`, `trust anchor` and `overhead`.

The measure refers to a method that is employed to mitigate integrity attacks on programs. Protection level indicates the level of abstraction at which protection is employed. Trust anchor specifies whether the measure relies on any root of trust, e.g. trusted hardware, or not. Finally, the overhead reports on the performance impact of a measure on the system.

1. **Measure.** A protection measure is the foundation of the protection activity. It can be done either completely locally (i.e. a program verifies its own integrity) or remotely (i.e. a trusted party remotely attests integrity). The inheritance relation between the measure,

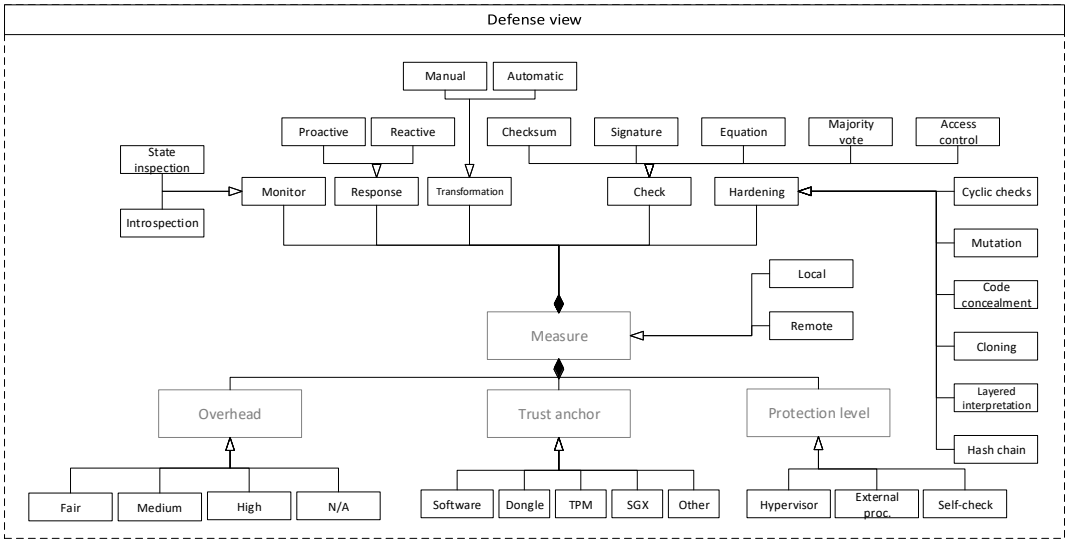


Figure 5: Defense view capturing the main classes, subclasses and relations of the defense dimension.

local and *remote verification* stands for this matter. Each measure itself is composed of five distinctive actions: *transform*, *monitor*, *check*, *response* and *harden*. These classes, although pursuing different objectives, contribute in protection a program. In the following, we describe the role of each action in the protection process.

(a) **Transform.** This action applies transformations on representations of assets. As mentioned earlier, these transformations can be carried out at different program lifecycle depending on the targeted representation and the artifact at hand. This indicates a link between the transform and lifecycle activity.

(b) **Monitor.** This component can actively inspect different representations of the system assets. Obviously, this component has to have a access to the system, otherwise it cannot audit anything. Hence, there is a tight binding between monitoring and representation. For monitoring system representations there are two approaches that are widely used in the literature, viz. introspection and state inspection.

i. **Introspection** monitors a set of static invariant properties of a program, e.g. code blocks, to detect tampering attacks.

ii. **State inspection** monitors the result/effect of the execution of a program, e.g. function return values, to reason about its integrity.

(c) **Check.** This component provides a mean to reason about the collected data by the monitor element, and decide whether assets are compromised or not. The output of this component, based on the employed technique, could be a binary (Yes/No) or accompanied with a confidence number. In the following we discuss the checking mechanisms that are commonly used in protection schemes.

i. **Checksum.** We classify any mathematical operation that converts large block of data into compact values, which is ideal for comparisons, as checksum based

techniques. CRC32 and hash functions fall under this category of checking mechanism.

ii. **Signature.** This refers to a cryptographic protocol for integrity verification by means of verifying the digital signature of artifacts.

iii. **Equation.** The result of a mathematical equation evaluation (with program runtime features) defines whether program integrity was violated or not.

iv. **Majority vote.** A set of functionally equivalent components disjointly carry out a computation and then collectively decide upon the integrity of the outcome.

v. **Access control.** A policy enforcement point beyond the control of the program and (possibly) attacker mediates the access to the critical resources.

(d) **Response.** Based on the check component's decision, the response component reacts in a punishing way to attackers. This reaction could vary from process termination, performance degradation, or even attempting to recover compromised assets. Technically speaking, there are two classes of response mechanism: *proactive* and *reactive*.

i. **Proactive.** Schemes utilizing this class of response act intrusively. That is, upon the report of integrity violating, immediate actions are taken to prevent attacks.

ii. **Reactive.** In some cases less intrusive and stealthier responses are desirable. In this model of schemes, the detection of integrity violation does not result in obvious reactions such as program terminations. A reactive response may, for instance, silently report on the violation without making the attacker or user realize.

(e) **Harden.** Since the measure itself can be subject to attacks, a consolidation technique is employed by the measure. For example a simple routine (say `monitor()`) that is solely responsible for auditing program state can easily get manipulated by an attacker. The same goes for response mechanisms. An open termination of the program as a response to tampers only adds a weak security.

Thus, it is crucial to add strength to protection measure using hardening techniques. Hardening aims to impede discovery process and thus raises the bar against attacks. This directly relates to the discovery and attack activities in the attacker view.

The hardening can be seen as adding a hard problem for the attacker to solve. This does not necessarily have to be a $np-complete$ problem, in some cases even a quadratic one will cause enough trouble for attackers and exhausts their resources, specially when the attacker is forced to manually solve the problem. For instance when an automated pattern matching fails to defeat the mechanism, attackers are doomed to manually analyze a large portion of the code, which, in turn, presumably deteriorates their success rates. In our literature review, we have identified six different hardening techniques that are commonly used by integrity protection measures. We discuss these techniques as follows.

i. **Cyclic checks.** In this model, protection is strengthened by using a network of overlapping protection nodes. Therefore, a particular asset (representation) may get inspected by more than one checker. In some variation of the cyclic checks, the checkers themselves are also protected by the very same mean.

ii. **Mutation.** Defeating a protection technique normally is the result of a process in which an attacker first has to acquire necessary knowledge about the scheme and its hardening problem, and then, utilizing the knowledge to break the protection. To this end, mutation techniques try to render attacker's prior knowledge irrelevant by frequently evolving the protection scheme.

iii. **Code concealment.** In the course of scheme analysis, attackers often refer to the program code and base their studies on it. Code concealment tries to impede this process by concealing a particular representation of the code. Nonetheless, program instructions have to get translated into legitimate machine code right before the execution. The later this translation occurs, the more an attacker is troubled. Some approaches tend to flush translated instructions after execution so at no point in time the attacker gets a chance to glance over to the entire application code at once. Program obfuscation and encryption are examples of this technique.

iv. **Cloning.** Multiple copies of sensitive parts are shipped in the application and at runtime a random predicate defines which clone shall be executed. This enables the protection scheme to remain partially functional even after successful attacks. In Section 5.8 we will discuss this measure in the context of concrete attacks.

v. **Layered Interpretation.** To utilize layered protection a program have to be run within an emulator or a virtual machine. Some schemes entirely virtualized the target application, while others virtualize some parts of the application. This technique enables employment of protection measures at a higher level of abstraction. Since programs has to be run by the host (hypervisor/emulator), they can be verified before execution and executed only if they pass the verifications.

vi. **Hash Chain.** Evolving keys and hash chain assures past events cannot be forged or forgotten. This technique is widely utilized to maintain an unforgeable evidence of the system state.

2. **Trust anchor.** In an abstract sense, this criterion specifies whether the scheme is entirely implemented in *software* or it is assisted by a trusted hardware module. Hardware based approaches are generally assumed to be harder and more costly to circumvent, due to the required equipments and knowledge at the attacker end. On the other hand, hardware based schemes add more cost, to acquire the required modules, for each client. In the following we discuss different trust anchors.

   (a) **Software.** Indicates that the scheme security is purely based on software hardening mechanisms without any trusted hardware whatsoever.

   (b) **Dongle.** Refers to hardware keys that could be used to store small chunks of data (e.g. keys) or program with more resilience against tampering attacks. None of the reviewed schemes in our survey utilizes dongles. Thus, we exclude it from our further classifications. However, according to [72] dongles are popular in practice.

   (c) **TPM.** Trusted platform module is a quite mature hardware chip that is used by plenty of hardware assisted integrity protection schemes.

   (d) **SGX.** Intel Software guard extension (SGX) offers dynamic process isolation to mitigate runtime integrity attacks.

   (e) **Other.** An indicator for reliance on hardware modules other than the explicitly stated ones.

3. **Protection level.** Indicates the enforcement of a protection scheme. It can happen (stated with inheritance relation) in three different abstraction levels:

   (a) **Self check.** This level is also known as internal protection in which a protection scheme is integrated into the program to be protected. Simply put, this process is in charge of carrying out its own integrity verifications and further decisions about how

to react to compromises. Applying this technique to a distributed architecture adds more resilience. Mainly because the protection schemes react disjointly, a single point of failure is prevented. On the negative side, however, all the decisions are solely based on the state of the program-to-protect, thereby these decisions may lack contextual information.

(b) **External process.** A dedicated process, Integrity Protection Process (IPP), monitors protected programs and responds accordingly. Unlike self-check approaches, IPP can combine multiple sources of contextual information to improve reasoning and thus responses. Also, it enables a mean for high level policy enforcement. These features, in turn, enhance both security and usability of a protection measure. However, there are two drawbacks in this model: **a)** it is challenging to capture the actual state of the programs via an external process, e.g. a malicious program can forge good states for IPP, and **b)** granted the high permission level of IPP, it can become a critical component to attack; single points of failure.

(c) **Hypervisor.** Protection is a part of a hypervisor logic. This entails that all processes are being virtualized and executed on top of the security hypervisor. The current state of the industry highly advocates this model. The hypervisor, in contrast to external process, does not have the problem of state forgery, as it (in theory) can capture all stealthy actions. Nevertheless, the problem of single point of failure remains as a concern.

4. **Overhead.** Performance overhead is an important aspect of integrity protection schemes. Practitioners indeed need an estimation of the overhead on their services. However, a great deal of the protection schemes have not been throughly evaluated, so performance bounds are unknown. We classified performance bounds into four classes as follows.

   (a) Fair($0 < overhead < 100\%$).

   (b) Medium($101\% < overhead < 200\%$).

   (c) High($overhead > 201\%$).

   (d) N/A. This class represents schemes with lack of experiment results or unjustifiable numbers due to limited experiments.

The defense view elaborates on techniques to protect integrity against attacks along with implications of such protections on system. Hence defense acts as a bridge between system and attack views. Moreover, defense view can be seen as a classification for integrity protection techniques, which can facilitate scheme selection by users.

3.4 View dependencies

Previously, we discussed all the three views of the taxonomy, their classes and dependencies in each view. In this section we provide an overview of the dependencies between the classes from different views. The high-level overview of these dependencies is depicted in Figure 6.

In the system view (in the middle), an association between the representation and asset (*"Contains"*) expresses the fact that system assets are exposed in different representations.

In the attacker view (on the right), the support of the tools in the whole process of reverse engineering is expressed by the link between the tools and reverse engineering (the *"Supports"* link). Attackers are the actor to execute an attack, this is also highlighted by an association between attacker and attack. The asset identification, which is the first objective of attackers, is depicted in form of the association between the discovery and representation classes (the

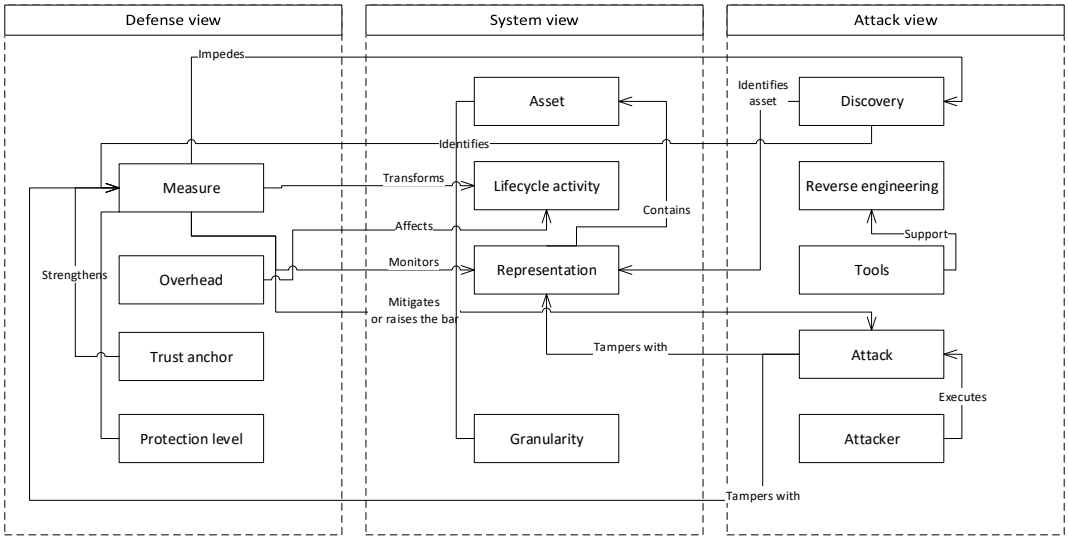


Figure 6: The high-level overview of the taxonomy showing the main classes of the defense, system and attack views along with their relations.

"Identifies assets" association). The second objective of attackers, after identifying assets, is to tamper with the representation of interest. This is captured by the link between the attack and representation (captioned with *"Tampers with"*).

The defense view (on the left) abstractly captures the defense process. A protection measure may rely on a trust anchor for additional strength, which is shown with the link captioned with *"Strengthens"*. In order to enable protection, a measure needs to transform a representation of the system in one (or more) of the life cycle activities. This is illustrated by the association between the measure and life cycle activity *("Transforms")*. Once a protection measure is in place, one (or multiple) representation of a system is actively or passively monitored, the *"Monitors"* association between the measure and representation shows this relation. Protection measures could potentially affect system performance and introduce overheads, which is shown with the *"affects"* link between the overhead and life cycle activity. Protection measures based on their hardening measure and response mechanism impede asset discovery and mitigate or raise the bar against attacks. The *"Impedes"* link from the measure to discovery (in the attack view) and *"Mitigates or raises the bar"* between the measure and attack depict these two interrelations.

However, protection measure is not the end of the game for attackers. They can target the protection mechanism itself, discover its logic and disable its protection. The *"Identifies"* link between the discovery and measure as well as *"Tampers with"* between the attack and measure captures this matter.

At this point we have covered all the three views of the taxonomy and their interrelations. For a complete view, Table 1 depicts the entire taxonomy. To support readability, for each element of the taxonomy we give an example in the context of sample DRM system. All the examples relate to the DRM system and thus we avoid repeating the context in each example.

Table 1: Software integrity protection taxonomy along with examples in the context of the sample DRM.

System view	Assets			Behavior	There are some sensitive logics that need to be integrity protected, e.g. license checker routine and stream viewer
				Data	Usage variables need to be protected against manipulation attacks. Otherwise, attackers can readily circumvent the usage policies
				D&B	The usage variables and any routine that has deals with these variables, in this case content provider, need to be protected
	Representation			Static	Malware or attacker may tamper with the DRM binaries to carry out their attacks
		Mem.		Code invar.	The static shape of the license checker code blocks in the memory is a target to attack for attackers
				Data invar.	Usage count variables in memory are the targets of tampering attacks
		Exe.		Trace	Tampering attacks causes identifiable changes in the program traces. For instance, dumping decrypted fragments in the stream viewer introduces additional calls in the trace
				Timed trace	Tampering with program routine will alter the program response time. For instance disabling license check routine by simply returning true could introduce new timing bounds
				HW counters	Hardware performance counters capture some technical details about the executing routine, such as number of branches, indirect calls and etc. Tampering with the license checker inevitably affects these numbers.
	Granularity			Instructions	All the individual instructions that access $client_{private}$ key need to be protected
				BB	The basic block in which license checking takes place need to be protected
				Function	The licenseChecker() and getExpiryDate() methods need to be protected
				Slice	The chain of instructions that read from or write to the usage count variables need to be protected
				Application	The entire stream viewer component (application) need to be protected
	Lifecycle a.			Pre-compile	License checker and data provider can be protected at the source code level
				Compile	License checker and data provider can be protected at the compiler level
				Post-compile	Stream viewer can only be protected in a post-compile process, because it is an external library
				Load	Every time the license checker is loaded in the memory a (new) protection is applied
				Run	The license checker routine is updated on the fly (during execution)
Attack view	Reverse engineering	Tools		Disassembly	Attackers may disassemble the DRM client to understand the logic
				Debugger	Attackers could inspect the license checking control flow using a debugger
				Tracer	Attackers could deepen their knowledge about the fragmented decryption in the stream viewer by dumping a trace of the program
				Emulator	An emulator could be utilized to facilitate program analysis
		Discovery		Taint analysis	Attacker can use tainting analysis to detect the checks that enforce usage policies
				Sym. exec.	Symbolic execution could be utilized to find instances of usage variables for which the program delivers the protected contents
				Memory split	Attackers could execute memory split attack to defeat self-checksumming based protections
				Patt. Match.	Attackers could use pattern matching to detect license checks, protection routines and usage policy enforcement points
		Attack		Binary	Attackers can tamper with the program executables at rest
				Proc. mem.	Attackers could leverge in-memory patching to carry out their attacks
				Run. data	Program input and output data is under control of the attackers.
				Ctrl flow	Attackers can subvert program control flow to for instance decrypt content without license check
Defense view	Meas.			Local	The DRM client is fully in charge of the integrity protection
				Remote	Security state of the DRM client is to be reported to the main server
	Mon.			State inspection	Monitor and check the results of the program execution
				Introspection	Read code pages of the license checker compare it to a known value
	Resp.			Proactive	Any detected attack raises the alarm and a delayed or an immediate punishing response is triggered
				Reactive	All accesses to the protected content (genuine or forged) is permitted, however, the server is notified about the violation of the usage policies in a postmortem verification. Server can add such users into a blacklist or file a lawsuit against them
	Trans.			Manual	Protectors have to execute manual tasks in the course of protecting a program, for instance, by developing non-trivial clones for the license checker
				Automatic	The protection transformation requires a limited user intervention and hence it is to a great extent automatic
	Check			Checksum	A checksum is computed over the code blocks of the license check
				Signature	Signature of the components are verified at runtime
				Equation eval	Access data forms a set of verifiable equation, e.g. total number of accesses is less than total number of views specified in the usage policy
				Majority vote	License checker is cloned and executed simultaneously by different threads/processes. Afterwards, a majority vote decides whether the license is valid or not
				Access control	All access requests need to pass through a master node which securely enforces usage policies
	Hardening			Cyclic checks	A network of integrity checkers collectively prevent software manipulation
				Mutation	The license checker code blocks are constantly modified at runtime to mitigate passive attacks
				Code conceal.	The License checker code is encrypted and only decrypted at runtime
				Cloning	The license checker is duplicated and every time at runtime a randomly chosen clone is executed
				Layered interp.	The client application utilizes a random instruction set that can only be executed on a custom secure virtual machine
				Hash chain	Usage event logs are securely chained to maintain an unforgeable evidence of the system usage. Such information could be used in postmortem verifications
	Overhead			Fair	Target clients have limited computation resources and thus a low overhead is desired
				Medium	Target clients have fairly good computation resources
				High	Sensitive parts of the program shall be protected strictly, high overheads are tolerated
				N/A	No information about overhead constraints
	T. anchor			TPM	Client systems do have a TPM chip
				SGX	Client systems support Intel SGX
				Other	Other trusted modules are decided to be used, e.g. a custom-built hardware
				Software	The protection is completely software based
	Prot. Lvl.			Internal	The client application is in charge of verifying its own integrity
				External	A dedicated integrity protecting process frequently verifies the integrity of client applications
				Hypervisor	A secure hypervisor is shipped with the client installation bundle. The client application can only be executed using the provided hypervisor

4 Applying the proposed taxonomy

We conducted a survey on integrity protection and tamper resistant techniques for software systems. This by no means is exhaustive, but we believe it serves as a representative of different techniques in the literature. Table 2 demonstrates the mapped literature on the proposed taxonomy.

Since there is no single research work that analyzes the resilience of protection schemes against MATE tools and discovery methods, we were not able to directly map the reviewed works to these criteria (in the attack dimension). To address this limitation, in the Sections 5.7 and 5.8 we reason about schemes' resilience based on their hardening methods. This enables us to first understand the role of hardening measures in the resilience of schemes and subsequently to complete the mapping. The space limitation hinders detailed discussion of the reviewed schemes. Instead, in the next section we correlate various aspects of the reviewed literature and discuss our findings.

5 Observation and Analysis

As part of our analysis we correlate interesting elements in the taxonomy from different views and discuss our findings. These correlations are those that we particularly find interesting for practitioners.

However, there are far more possible correlations that one can carry out. To address this concern, we published a web-based tool that enables end users to correlate any arbitrary pair of elements from different dimensions. The tool is accessible at `http://www22.in.tum.de/tools/integrity-taxonomy/`. In the following we report on the dependencies of the taxonomy dimensions.

5.1 Asset protection in different representations

We defined integrity as a property of software that applies to both data and behavior. From the user perspective, it is a daunting task to find protection schemes that protect data and/or logic at a particular representation.

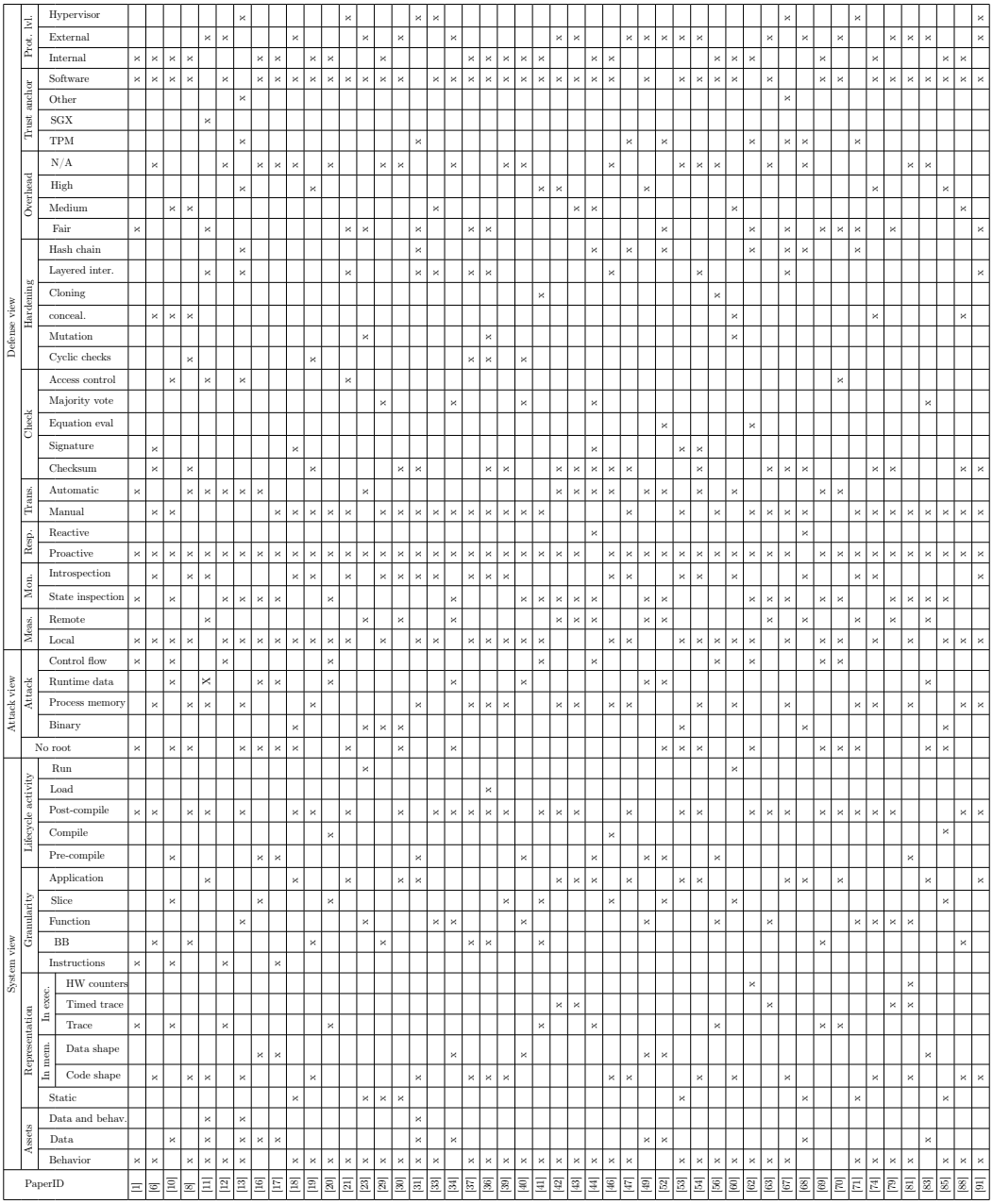
Signature verification is one of the techniques that users commonly utilize to protect integrity. The static signature verification is natively supported by almost all operating systems. These schemes are rather a non-preventive security measure to protect users, but not softwares. Clearly, this does not match the MATE attacker model, where the user is also the attacker.

Besides, these techniques have two shortcomings: **(i)** their security relies on operating system settings, which could be easily manipulated, and **(ii)** they suffer from the *Time of Check–Time of Use (ToCToU)* limitation in which signatures are verified on the static representation of programs upon execution, while many attacks (including in-memory patching) could potentially occur after execution is started. [48] reported that over 60% of attacks published by CERT were either buffer overflows or of ToCToU attacks.

Due to the aforementioned limitations, the static signature verification is not an optimal technique as it fails to protect other asset representations during execution.

A natural question that may come to ones mind is why protection for a certain representation instead of protecting assets in all the representations. The answer is threefold. First, attackers might not have the right privileges on some representations, e.g. accessing in-execution representation of processes requires *root* privilege, except when attackers find a way to inject their attacks into the program of interest. Reasonably, to avoid unnecessary overhead we aim to protect the representations that are at risk. Second, some risks might be acceptable for a certain use case,

Table 2: Applied taxonomy on the surveyed literature.



e.g. in IoT programs naively protecting static representation of a program might be sufficient, given that runtime protection imposes unacceptable overhead. Finally, 100% protection of assets in all representations is technically infeasible. Therefore, to shed light on this concern, we correlate software integrity assets to different representations in order to identify corresponding protection schemes.

The desired correlation is depicted in Table 3. In the following we discuss the relevant schemes according to their asset-to-protect.

Table 3: The correlation between the integrity asset protection and representation.

		Integrity assets		
		Behavior	Data	Data and behavior
Representation	Static	[30] [29] [71] [85] [23] [53] [18]	[68]	-
	Code inv.	[31] [39] [60] [13] [67] [8] [47] [37] [46] [19] [74] [6] [81] [36] [54] [88] [91] [11]	[31] [13] [11]	[31] [13] [11]
	Data inv.	[40]	[49] [16] [83] [52] [34] [17]	-
	HWC	[62] [81]	-	-
	Timed trc.	[63] [42] [79] [81] [43]	-	-
	Trace	[41] [1] [20] [44] [56] [12]	[10]	-

5.1.1 Behavior protection schemes

As the correlation suggests, a majority of behavior protection schemes with a total number of 18 [31, 39, 60, 13, 67, 8, 47, 37, 46, 19, 74, 6, 81, 36, 54, 88, 91, 11] target the *code invariants* representation in their protection mechanism.

Meanwhile, [40] referred to the shape (invariants) of computed data to reason about the behavior integrity. [62, 81] took hardware performance counters (e.g. the number of indirect calls and elapsed times) into account to evaluate programs' integrity.

Some efforts were found in the literature to raise the bar against program manipulation when programs are at rest (binary). [30, 85, 18] are examples of hardening binaries by statically analyzing them and subsequently adding resilience against tampering attacks. [29] rather presented a theoretical concept to design more resilient integrity protection schemes. [71, 53, 23] proposed techniques to protect program binaries after distribution.

In order to manifest some tampering attacks, attackers inevitably need to change, inject or reorder instructions. In this situation, the order in which program instructions are being executed along with the data that they read from or write to, i.e. a program trace, is a source of knowledge to detect such attacks. This information could be extracted form a program's execution trace. [20, 41] incorporated memory accesses and branching conditions in the program trace into a hash variable to later match it against the expected trace hash. [44] passively verified the execution traces that are reflected into log records. [56] duplicated sensitive regions in the program to add more resilience to attacks by forcing perpetrators to tamper with all clones of sensitive codes.

The correlation also indicates that timing analysis (timed trace) is utilized in [63, 1, 62, 42, 79, 81, 43, 12] to verify integrity of the program behavior. The assumption in the mentioned schemes is that a certain untampered routine expresses identifiable execution time characteristics.

5.1.2 Data protection schemes

Tampering with a program's data can enable attackers to manifest integrity violating attacks. The simplest way to do so is by attaching a debugger and subverting the program flow, for instance, by flipping a register's value. As the correlation suggests, several schemes protect the program's data integrity. [49] introduced a hash-chain alike concept for data integrity with the

help of a semi trusted entity. [83] equipped system data flow analyzer nodes and routed the traffic to them for integrity analysis. [34] proposed to duplicate sensitive processes and compared their computed values based on a majority vote scheme. [17] used data flow graph to verify whether the application conforms to the genuine data flow model at runtime. [52] collected data invariants in a program and verified them at runtime. [20] computed a cumulative hash of memory accesses at the instruction level to form a hash trace of the program execution.

5.1.3 Data and behavior protection schemes

Protecting data and behavior together might be a more appealing option when the program to protect possesses both sensitive data and logic. In our survey, we found three schemes that aim for data and logic protection. [31] proposed a technique in which a program's logic as well as data are isolated from external processes by a secure hypervisor. They also designed a process to encrypt data before persistence, so that the data is never exposed to adversaries in plain text. [13] in addition to logic and data isolation, introduced a secure inter-process data propagation in a (nearly) real time manner. [11] designed a fully isolated execution environment for the process as a whole on Intel SGX hardware commodity. All the three schemes rely on trusted hardware.

The presented correlation singles out relevant schemes for protecting different integrity assets (behavior, data and data and behavior) in different representations. This facilitates the scheme selection based on the assets and representations at the user end.

5.2 MATE attack mitigation on different asset representations

A program possesses different representations depending on its state, viz. static, in-memory and in-execution. Each of theses representations are subject to tampering attacks. Protections on representations closer to the time of use (i.e. execution) could prevent or detect a wider range of tampering attacks. That is, the static representation of a program is implicitly protected, if the very same program utilizes in-memory protections.

In this paper, we limit our adversary model, regardless of how attacks are executed, to four generic attack goals: binary patching, process memory patching, runtime data modification and control flow hijacking. Constrained by their permissions, attackers can choose different program representations as their attack targets.

In this correlation we aim to identify protection techniques that mitigate the aforementioned generic attacks on different representations. For this purpose, we relate representations in the system view to attacks in the attack view. This is depicted in Table 4. We discuss our findings classified by attacks, viz. binary, control flow and process memory as follows.

Table 4: Defense mechanisms against tampering attacks for different (asset) representations.

		Representation					
		Code inv.	Data inv.	HWC	Static	Timed trace	Trace
Tampering attacks	Binary	-	-	-	[68] [30] [29] [85] [23] [53] [18]	-	-
	Control flow	-	-	[62]	-	-	[10] [1] [41] [69] [20] [44] [56] [70] [12]
	Process memory	[31] [39] [60] [13] [67] [8] [47] [37] [46] [19] [74] [6] [81] [36] [54] [88] [91] [11]	-	[81]	[71]	[42] [81] [43]	-
	Runtime data	-	[49] [16] [83] [40] [52] [34] [17]	-	-	-	[10] [20]

5.2.1 Binary

Static manipulation (aka static patching) targets a program while it is at rest (prior to execution). To prevent these sort of attacks the surveyed literature suggests static representation verification as well as a set of hardening measures to raise the bar against tampering attacks in the host. Note that we do not include signature verifications supported by operating systems, as they do not target MATE attackers. In the following we review the schemes that protect the static representation of programs.

Static representation verification [53] and [30] proposed remote file integrity protection using file system hashing. The main difference lies in the fact that the latter initiates the hash with a remote challenge. However, this technique can easily be defeated if the attacker keeps a copy of the genuine files just for the sake of hash computations.

Hardening measures Neisse et al. [68] suggested to plug (hardware-assisted) configuration trackers in the system to monitor modifications on the file system. All modifications are then signed by a TPM within the host and then reported to a third party verifier to judge about the maliciousness of the actions.

[18] designed a protection scheme to defend against dependency injecting and tampers with the update scripts in a system by utilizing dynamic signature verification and system call interposition in the kernel. They argued these two can mitigate the persistence of a compromise in the system. Meanwhile, [29] proposed to strongly connect integrity protection code with the normal program control flow and discussed theoretical complexity using a graph based game.

To add resilience against dynamic attacks such as buffer overflows in the static representation, [85] proposed an efficient means to detect potential integrity pitfalls in a distributed system by first merging all network wired nodes into a unified control flow graph, so called *Distributed Control Flow Graph DCFG* and then running a static analyzer to find all user-input reachable buffer overflows. In their way, intuitively, pitfalls can be identified with less false positives as opposed to analyzing each node individually.

Collberg et. al in [23] proposed to protect client programs' integrity by forcing constant and frequent updates. In their approach, a trusted server constantly uses software diversification techniques to change a server's method signatures, which in turn coerces the client to update all the program artifacts right after each mutation.

5.2.2 Control flow

As the correlation suggests, there are two representations that were utilized to prevent a program's control flow manipulation, viz. HW counters and traces.

Hardware performance counters were used in [62] to reason about the program's control flow integrity based on performance factors such as the number of calls and the elapsed time on each branch.

The trace representation was used by 9 protection schemes. These schemes are listed as follows. Banescu et al. in [10] proposed a technique to ensure only genuine internal calls can access assets. This is done via a runtime integrity monitoring that traces stack's return addresses and compare them to a white list of call traces. Stack inspection works well for synchronous calls, however, due to the incomplete trace information, trigger and forget calls cannot be handled in this way. To cope with asynchronous calls, caller threads are verified using a Message Authentication Code (MAC mechanism).

Abadi et al. in [1] proposed a scheme to protect the integrity of programs' control flow by injecting a set of caller reachability assertions in the beginning of program branches.

Chen et al. in [20] proposed to compute a checksum over a program's execution trace by hashing its assignment and branching instructions. These hashes are later verified at desired locations in the protected program. Similarly, [41] proposed to compute a hash over a program's assignments and executed branches. These hashes later are used to dynamically jump to the right memory locations. Mismatches in hashes crash a program as they potentially cause jumps to illegitimate addresses. Dynamic jumps are created by the mean of interleaving multiple basic blocks in super blocks with multiple entry and exit points. In effect, the hash values determine which entry/exit points the program shall take at runtime.

To capture an unforgeable evidence of the traversed program's control flow, Jin et al. in [44] proposed a protection mechanism based on secure logging (forward integrity). Their scheme is detective. That is, tampering attacks are detected in a postmortem analysis after they took place. [56] mitigated call graph manipulation by utilizing opaque predicates to randomly switch between a set of nontrivial clones of sensitive program logic (input by human experts). [70] designed a mitigation for ROP attacks by monitoring branching behavior and size of fragments executed before each branch. [12] proposed a mechanism in which a program-to-protect subscribes to an external process which dictates the genuine control flow of the application. The external process can be protected by other expensive integrity protections without introducing extensive overhead on the main application.

5.2.3 Process memory

Protecting process memory based on *code invariants* is the most common approach in the literature. In the following, we briefly introduce different schemes based on this technique.

Self-encryption is a defense mechanism utilized by a number of protection schemes. Aucsmith [6] proposed a scheme that utilizes code block encryption and signature matching to mitigate tampering attacks on process memory. The author refers to the integrity checking codes as *Integrity Verification Kernel (IVK)*. These IVKs, for better stealth, are interleaved with the program functionality. Using this technique, the entire application code except for the starting block is encrypted. From the memory layout of each executed block a key is derived which can decrypt the next block. [88] is another encryption based protection in which the key to decrypt the next basic block is derived from a chain of hashes of previous blocks. Thus, tampering with blocks breaks the chain and results in an unrecoverable program. [74] applied the self-encryption with re-encryption after each invocation to protect Android intermediate code. Protected programs using this scheme, however, cannot use reflection nor host service contracts (otherwise unreachable) due to the code encryption.

Self-hashing is another mechanism that is employed by a number of process memory protection schemes. [81] used reflection to obtain program codes at runtime. Program codes are subsequently verified using a hash function. In addition to hash verification, HW counters are also utilized to increase the resilience. Chang and Atallah in [19] proposed a software protection technique that builds up a network of compact protection blocks (aka Guards) that along with performing integrity checking of the application blocks (basic blocks) can also verify other protection guards. These protection guards, provided that an untampered version of the code is available, can potentially heal the tampered regions of the program. [8] proposed a technique that utilizes a network of cyclic checkers with multiple hash functions to prevent memory tampering attacks. This scheme can cope with the relocation problem in binaries and hence it protects instructions that contain absolute addresses. To the best of our knowledge, this scheme

is the only self-checksumming protection measure whose source code is publicly available. [39] proposed a check and fix protection scheme that utilizes a set of linear testers that compute a hash over certain regions in the process memory. Upon mismatch detection correctors attempt to heal the tampered program.

[37] proposed a technique combining cyclic checks and self-encryption. In this scheme, first cyclic guards are injected and then the entire program is encrypted. An emulator is shipped with the program which can decrypt and eventually execute it. The decryption key is shipped into the binary using white-box cryptography, so attackers can not easily extract it from the program. [36] did a similar protection with the difference of using virtualization obfuscation instead of self-encryption. In addition to that, a set of self-checking guards are generated on the fly to protect the dynamically translated program in the cache. [46] applied control flow flattening obfuscation transformation in combination with self-hashing protection guards. In their scheme, random intervals of a program are checked using CRC32 checksums.

[60] used constant mutation at runtime so that the same memory address is used for multiple code and data blocks in one execution.

Secure hypervisor is used by some protection mechanisms to protect the process memory. [31] used a secure hypervisor to verify code pages of the protected programs. [54] proposed a measure that utilizes a secure emulator, code singing and encryption. First programs code blocks are signed and encrypted. Later, the secure emulator can decrypt and execute the genuine code blocks. [91] proposed a hypervisor monitoring tool called VMI to introspect the memory of running virtual machines. VMI has to, however, be executed with root permission on the host. [13] aimed for integrity protection in IoT devices by proposing a concept of secure (tamper resistant) tasks enabled by a trusted hardware. Furthermore, a secure interprocess communication protocol is also designed. In order to pass a secure message, a source process loads the message m and destination id, d_{id}, into CPU registers and then triggers a secure IPC interrupt. The interrupt eventually makes m along with the s_{id} accessible for destination process/task. [67] proposed a hardware assisted hypervisor that enables users to define their custom integrity routines. Similarly, [47] proposed to equip hosts with custom attestation and measurement probes to detect violation of integrity.

[11] proposed a technique to use SGX secure enclaves to place the entire application in isolated containers. This prevents attackers form runtime process memory manipulations.

Static protection to isolate process memory was done in [71]. Their goal was to ensure genuine geolocation information by starting the service in a lightweight hypervisor which is secured by a TPM.

Timed traces are another representation which was used to build protection schemes. In effect, [43, 42] based their schemes completely on timing analysis to verify process memory. In this approach, once the attestation is launched, all active memory contents are copied to the flash memory. Then, the verifier sends a random seed that the attestation client uses for proof of computation. At different check-points the verifier sends further seeds and defines the memory addresses that need to be incorporated in the proof computation. Meanwhile, timing analysis could be used as an additional source of information. For instance, [81] used reflection as its core of integrity protection, but also utilized timing analysis for additional hardening.

5.2.4 Runtime data

To protect runtime data, data invariants and trace representations were utilized by the reviewed protection schemes. The correlation indicates that 7 schemes operate based on the data shape representation. [49] relied on an authenticated data structure (Merkle tree) to enable users

[0]https://github.com/google/syzygy/tree/integrity

to verify correctness and freshness of the access control policies and user data. [16] used a combination of static analyzer and memory analyzer to detect data tampering that leads to kernel exploits. [83] used SDN to route data to an extra node to verify network data integrity. [40] proposed to utilize a set of checkers are added to the program to check return values of the sensitive functions. Inputs for these checkers are generated using symbolic execution. [34], similar to [40], proposed to feed the return values of a given service to a set of clones and have a majority vote based scheme to decide upon integrity of the service. [52] proposed to first run a program using the Daikon tool [32] to capture dynamic data invariants. From these invariants later a set of guards are generated and injected into the program of interest. At runtime these invariant should hold, otherwise response function is triggered. [17] proposed a technique which uses data flow graph to build a data model. At runtime the conformance of the application data flow to the model is frequently verified.

Trace representation was used in two protection schemes to authenticate data integrity. [10] utilized white box cryptography to protect the integrity of a configuration file in the Chromium browser. A white box proxy enables Chromium to sign and verify genuine configuration files. While attackers have difficulties extracting the key from white box crypto, they might be able to subvert the execution flow and misuse the proxy to sign malicious configuration files. Therefore, this scheme authenticates any call to the proxy.
[20] incorporated a selected set of memory references into hash variables. The constant program data can be protected using this measure. However, incorporating input data will make it impossible to precompute expected hashes, due to nondeterminism in programs.

In this correlation we identified and discussed the different schemes to mitigate particular MATE attacks, viz. binary, control flow, process memory and runtime data manipulations, at different representations. Amongst the reviewed schemes only [85] targeted distributed systems. This work, however, focused on adding resilience to the static representation against buffer overflows. All the other reviewed schemes focused on protecting assets in a centralized application, which might be in contact with a remote server. However, these schemes clearly fall short in protecting a software system that is comprised of a set of distributed nodes, communicating over network. Protecting program representations in a distributed architecture appears to be a major gap.

5.3 Defense integration to program lifecycle

Each defense comes with a mean in which monitoring probes are hooked, connections to response algorithm are established and hardening is applied. In a defense mechanism *Transformation* is the component that employs protection in a program by applying necessary modifications. Transformations may target different stages of a program varying from source codes to executable binaries. This will impose some constraints on the applicability of defenses to different contexts. For instance, one may have no access to the source code for a third party component. That is, for such cases, all the defenses based on source code transformations become irrelevant. The stream viewer in the DRM sample is an example of such components.

In addition to the constrains imposed by lifecycle activities, program representations on which protections are applied impose further constrains. For instance, monitoring memory can impose intolerable overheads in IoT devices. Therefore, it is of major importance to identify integration constraints of protection schemes in different application contexts. To capture this concern, we analyze the correlation between transformation target and system representation. Table 5 illustrates the outcome of this correlation. We discuss the correlation by iterating over lifecycle activities as follows.

Table 5: Defense mechanisms for different (asset) representations and their correlation to program lifecycles.

		Representation					
		Static	Code inv.	Data inv.	HWC	Trace	Timed trace
Lifecycle	Pre-compile	-	[31] [81]	[49] [16] [40] [52] [17]	[81]	[10] [44] [56]	[81]
	Compile	[85]	[46]	-	-	[20]	-
	Post-compile	[30] [71] [53] [18]	[39] [13] [67] [8] [37] [19] [47] [74] [6] [36] [54] [88] [91] [11]	[34]	[62]	[1] [41] [69] [70]	[63] [42] [79] [43]
	Load	-	[36]	-	-	-	-
	Run	[23]	[60]	-	-	-	-

5.3.1 Pre-compile

In the following we have a closer look at the schemes that operate on the source code level and discuss their correlation to the asset representations.

Code invariants [81] proposed a scheme for remote verification of integrity protection. The idea is to retrieve a program's code shape using reflection and subsequently computing a hash over it. This hash is then reported to the remote party. These reflection calls are injected into the program source code.

[31] proposed a scheme that requires developers to generate integrity manifests for all programs-to-protect. Integrity manifest contain program measurements and a unique program identity. These manifests need to be shipped to a secure hypervisor which follows the measurements to evaluate the integrity of programs at runtime.

Data invariants According to the correlation outcome there are four schemes that operate on data shape to verify software integrity. We will discuss these schemes, focusing on their integrability to systems. Data invariants verification is the technique that is used in [52]. In order to capture these invariants, the *Daikon* tool is utilized, which requires programs to be instrumented and subsequently executed. Afterwards, in the protection phase, these invariants are verified by a set of protection guards.

[49] proposed a mechanism to protect program inputs using a set of data integrity polices specified by users. These policies are enforced by the mean of an authenticated data structure [65].

In order to protect program functionality at runtime, [40] proposed a scheme for C# programs in which return values of program functions are tested in a network of cyclic checks. To construct checkers, the PEX symbolic execution tool [86] is used. PEX generates test cases (input-output pairs) for the functions of interest. Checkers are generated based on the test cases. They invoke a protected function with the test's input arguments and subsequently match the output of the function with the expected result of the test case.

Conversely, [16] designed a scheme that inspects kernel's dynamic data to detect tampering attacks. They claim that their scheme can protect up to 99% of such attacks on kernel.

Trace The following schemes rely on trace data to evaluate program integrity. Given that log records intrinsically capture enough information about the executed trace, [44] based their scheme on recovering a program trace from its logs. In this scheme program code needs to be augmented with a comprehensive logging mechanism. A hash chain mechanism is utilized to

preserve the forward security properties. The logs are supposed to be verified by a trusted external entity Since attackers cannot forge logs, the verifier can detect malicious activities during verifications. [56] proposed a scheme which aims at diversifying program's execution trace by randomly executing non-trivial clones of the sensitive functions. The downside is that all the non-trivial clones have to manually be implemented. [10] aimed at defeating control-flow hijackers by a set of stack trace introspection guards. These guards are injected in the prologue of sensitive functions in the source code.

5.3.2 Compile

Compiler level protection could potentially generate far more optimized protection routines thanks to built-in optimizations. Despite the benefits, compiler-based protections require access to the source code. Also, compiler transformations will target a specific compiler and hence restrict the choice of compilers at the user end.

Furthermore, since compiler optimization passes are oblivious to protections implemented, they could potentially break them and thus cause false alarms in protected programs. For instance, some compilers utilize enhanced memory caching by block reordering. This relocates program blocks after protection is laid out and hence breaks the hash checks in self-checksumming based protections, resulting in false alarms [8].

In our survey we only found three compiler based protections. Junod et al. [46] implemented a scheme that aims at protecting *code shape* by a combination of control flow flattening and tampering protection in a compiler pass (implemented in the LLVM infrastructure). On the negative side, no evaluation results on the tamper resistance overhead were published in this work. In an effort for securing distributed programs against buffer overflow exploits, [85] proposed a LLVM based analysis tool that combines the CFG of all individual programs to construct a distributed CFG. The distributed CFG is to improve the accuracy of vulnerability detection. Oblivious hashing [20] is another scheme that is implemented in abstract syntax trees. It operates on in-execution representation to authenticate program *trace*. In this transformation, program assignments and branching conditions are reflected in a hash variable that serves as a trace hash. Later, this hash could be verified at any desired point in the program.

5.3.3 Load

In our survey, we only found one research work that transforms program instructions at load time. Ghosh et al. in [36] proposed a tamperproofing technique on top of instruction virtualization obfuscation[1]. In this scheme, a mechanism was designed to protect translated instructions (code invariants) within the emulator in a program by safeguarding them using a network of checkers (similar to [19]) along with frequent cache flushes.

5.3.4 Run

Transforming applications at runtime is another option. These transformations render adversaries' knowledge obsolete by turning the protection into a moving target. In this setting, attackers have a limited time before the next mutation takes place, and hence their success rates deteriorate. Collberg et al. [23] proposed a scheme for client-server applications in which the server API is constantly mutated using diversification obfuscation techniques, forcing clients to

[1] It is a technique in which program instructions are randomized such that only the shipped emulator can execute them [5]

update frequently. Assuming that the server is secure, attackers have a limited time to carry out reverse engineering attacks on client applications (*static*). Soon after the server API is mutated, client applications have to be updated to reflect API usage changes, which renders attackers knowledge obsolete.

The downside of this scheme, however, is the obligation of having the server and the clients constantly connected to receive highly frequent updates

Similarly, [60] proposed a tamper resistant scheme based on *dynamic code mutation* in which program's data and code in the process memory are regularly relocated such that code and data memory cells are indistinguishable. That is, a particular memory cell (which is allocated for the process memory) serves both data and code during the course of an execution. Dynamic code mutation aims for keeping the code unreadable until execution time. Therefore, we classified this scheme as *code invariants* protector.

The basic assumptions in this technique are that reverse engineering (and hence tampering with programs) is harder **a)** when program code and data are indistinguishable, and **b)** when there is no fixed mapping between instruction and memory cells so that the same memory region is used by multiple code blocks at a course of execution.

The general observation is that runtime transformations are technically much more challenging to implement, which justifies the limited number of schemes in this category.

5.3.5 Post-compile

Post compilation transformation operates at the binary level. That is, compiled programs and libraries, without presence of their source code, can be protected. Because of their applicability on majority of product line, plenty of schemes utilize post-compile transformation. In fact, we found protection schemes for all the system representations which carry out their transformations in a post-compile process. In the following, we elaborate on these schemes classified by their target representations.

Code invariants Schemes that protect the code invariants (introspection) in a post-compilation transformation process are listed in the following. To protect integrity of Android applications [74] proposed a self-encrypting scheme that operates on the intermediate representation of Android programs, i.e. Dalvik executable files. Although the key is shipped with the program, this scheme adds resilience against tampering attacks. [8] applied self-checksumming transformation on windows portable executable (PE) binaries. [19] used cyclic network of checkers with repairers to recover tampered blocks to protect WIN32 (DLL and EXE) binaries. Similarly, [39] protected binaries based on self-checksumming technique. [47, 67, 91, 54, 13] proposed schemes that operate at the hypervisor level to protect program binaries. [11] used an isolation supported by Intel SGX to place the entire applications inside protected enclaves and protect them from attackers. [37, 54, 6, 88] proposed to encrypt program binaries in order to protect it. [36] designed a method that transforms program binary into a random instruction set to carry out protection.

Data invariants We identified two schemes that protect data integrity in after compilation. In the scheme proposed by [34] multiple clones of a program are simultaneously executed to detect instances that diverge from the majority of the clones. Malone et al. in [62] proposed a scheme that has two phases: *calibration* and *protection*. In the first phase a program is executed while hardware performance counter (HPC) values are continuously collected. A mathematical tool is then utilized to identify hidden relations among the gathered data in form of equations, which essentially captures the effect of genuine execution of the program on HPC. Finally, a set of guards are injected in the program to evaluate those equations at different intervals. However,

both Windows and Linux operating systems prohibit any access to HPC instructions in user-mode. This essentially requires applications to invoke a kernel call to read HPC values at runtime, which opens the door for call interceptions.

Static (file) To protect the integrity of static files checksum based techniques were proposed in the literature. [53] introduced a tool to verify signatures of a large number of binaries on a system. [30] enabled a remote verifier to compute and verify checksums of static files residing on a server. In this model, the verifier sends a challenge with which the server has to initialize checksum variables. To protect operating system static services [18] proposed a technique to guarantee the integrity of standard operating system application by safeguarding core software package providers.

As a mean to protect specific services in a system, [71] developed a genuine geolocation provider service that is enclosed in a statically secured lightweight hypervisor.

Timed trace [79, 63] are two consecutive schemes that authenticated the integrity of legacy systems (no multi core CPU) by measuring the time differences between genuine and forged execution of programs (timed traces). Similarly, [42, 43] also relied on timing analysis to detect malicious behaviors in remote programs. In these schemes, the verifier requests a checksum over certain regions of the process memory (memory printing) and at the same time measures the elapsed time. The checksum value and timing analysis enables the verifier to reason about integrity of the system. Nevertheless, during the memory printing process the system must stop functioning, otherwise the timing data will be inconsistent.

Trace Trace representation is utilized by a set of protection schemes to protect control flow integrity. Abadi et. al in [1] proposed a scheme in which the integrity of control flow is verified by a simple token matching scheme. For this purpose, before every jump instruction a unique token is pushed into the stack. Subsequently, at all destinations (jump targets) a predicate is added to verify the token which should have been previously added to the stack. [41] leveraged invariant instruction length on x86 architecture to design a scheme with which tampering leads to an inevitable crash at runtime. [70] verified branching patterns to detect ROP attacks at the Windows 7 kernel, which can transparently protect all executing applications without any modification in them.

In the correlation of transformation lifecycle and protected representations, we classified schemes based on their representation-to-protect and their applicability on different program lifecycles. Our results indicates a gap in compiler based, load and run time protection schemes.

5.4 Correlation of transformation lifecycle and protection overhead

Another interesting direction to look into is the influence of the transformation lifecycle on the overhead of schemes. Compiler-level transformations leverage optimization passes in the compilation pipe line. Naturally, applying transformation at compiler-level should impose less overhead in comparison to post-compile transformation schemes. Mainly because binary level modifications occur at the end of the build pipe line, where no further optimizations are carried out. To capture this view we correlate overhead classes with the transformation lifecycles that is presented in Table 6. We map our surveyed papers on 4 classes of overhead indicators, viz. N/A, Fair, Medium and High.

Table 6: The correlation between the transformation lifecycle and overhead.

		Lifecycle				
		Pre-compile	Compile	Post-compile	Load	Run
Overhead	Fair	[31] [52]	-	[1] [62] [21] [71] [67] [69] [37] [79] [36] [70] [91] [11]	[36]	[23]
	Medium	[10] [44]	-	[33] [8] [88] [43]	-	[60]
	High	[49]	[85]	[41] [13] [42] [19] [74]	-	-
	N/A	[56] [16] [81] [40] [17]	[20] [46]	[30] [39] [63] [6] [54] [34] [53] [18]	-	-

5.4.1 Fair

As can be seen in Table 6, a majority of schemes with fair overhead are directly applied on the binaries, i.e. post-compile transformation. Precisely speaking, [36] with 10% overhead (in addition to the virtualization overhead which is reported to be about 30%) from the load category, [31, 52] with 8-10% overhead from the pre-compile category, and [23] with 4-23% overhead from the run category.

5.4.2 Medium

In our survey we identified 6 schemes with a medium overhead. In post-compile category [33] with 128%, [8] with 134% for non-CPU intensive applications and [88] with 107% are classified as schemes with a medium overhead. In the pre-compile category, [10] with 130% and [44] are in medium class. In the run category, [60] with an overhead of 107% (for a program with 70 protected basic blocks) falls under this overhead class.

5.4.3 High

In the high overhead class, we found 7 schemes for pre-compile and post-compile transformation activities. [49] imposes 5x slowdowns on reads and 8x slowdowns on writes. Similarly, [74] introduces an overhead of 500%, [41] and [13] both are reported to impose 3x slowdown in protected applications. [85] utilizes an algorithm with complexity of $O(N^2 + 2^N)$.

5.4.4 N/A

We mapped all the schemes without performance evaluations or insufficient results into this class. A major problem with the performance evaluation of the reviewed protection schemes is lack of comprehensiveness. Thus, we are incapable of estimating the overhead of over 16 protection schemes. Surprisingly, the two compiler-level protection schemes [46, 20] also fall under this category.

This correlation once again shows that there is a gap in benchmarking protection schemes. To the extent that from our results we cannot approve nor reject the hypothesis that compiler-level transformations perform better.

We would like to emphasize here that this classification is just an estimation based on the authors' claim on the efficiency of their proposed schemes. Therefore, depending on what dataset was used in the evaluation process by the authors, the overhead might differ. A fair evaluation would be to use a constant dataset to measure overhead of different schemes.

Unfortunately, this is not possible due to two reasons. First, to the best of our knowledge, very few of the reviewed schemes were made open source. Secondly, even if the source codes were available we would not be able to compare their performance without further classifications.

Because these schemes are designed for different operating systems and different representations. Thus, they need to be clustered technologically and then evaluated.

5.5 Check and Monitor representation

Protection schemes, as stated earlier, operate on a particular representation of the assets. This implies a sort of monitoring mechanism on the representation of interest. The collected data (in the monitoring process) need to be analyzed and ultimately verified by the scheme's *Check* compartment. Studying the different check methods for verifying different asset representations is particularly interesting. Table 7 depicts the correlation of check and representation items. Based on this correlation we report on different check methods that were used in the literature as follows.

5.5.1 Code invariants

Expectedly, the majority of code shape verifiers with a total number of 13 use checksum based techniques, which also includes hash functions In addition to checksums, [6, 54] employed signature matching as a secondary measure for tamper detection. [11, 13] maintained code invariants security by enforcing access control at a trusted hardware level.

5.5.2 Data invariants

In order to check the data shape, 3 schemes [83, 40, 34] used majority vote principle. On the other hand, [62, 52] used equation system to verify the integrity.

5.5.3 Hardware counters

[62, 82] used hardware performance counters in their checking mechanism.

5.5.4 Timed trace

[42, 43, 63, 79] computed a checksum alike routine and measure the elapsed time.

Table 7: The correlation of the representation and check mechanism.

		Asset representation					
		Static	Code inv.	Data inv.	HWC	Timed trace	Trace
Check mechanism	Access control	-	[13] [11]	-	-	-	[10] [70]
	Checksum	[68] [30]	[31] [39] [67] [8] [47] [46] [19] [74] [6] [36] [54] [88] [91]	-	-	[63] [42] [79] [43]	[44]
	Equation eval	-	-	[52]	[62]	-	-
	Majority vote	[29]	-	[83] [40] [34]	-	-	[44]
	Signature	[53] [18]	[6] [54]	-	-	-	[44]

The correlation between checking mechanism in different protection schemes suggests that checksumming is the most used technique for integrity verification. Other checking measures such as signature verification, access control, equation evaluation and majority votes are equally unpopular.

5.6 Monitoring representation correlation with protection level (enforcement level)

In this subsection we are aiming to find out which representations could be monitored at which protection level. For this matter, we correlate *protection level* and *representation* in Table 8. Since we are interested in identifying the possibility of monitoring representations at different levels, we only indicate the number of citations instead of citing the references.

Table 8: The correlation of the Representation and protection level.

	Static	Code invariants	Data invariants	HW counters	Timed trace	Trace
Internal	2	10	3	1	-	7
External	5	5	4	1	5	2
Hypervisor	1	4	-	-	-	-

The correlation indicates that both the internal and external protection levels have no limitations on monitoring system representations, with the exception of traces which were not targeted by any of external protectors in the reviewed literature.

Furthermore, the correlation shows a gap in the hypervisor-based protection techniques in which none of the data shape, hardware counters and traces were used in the reviewed schemes. It is not clear whether this is due to a technical limitation of hypervisors or simply a research gap. Further studies should be conducted to address this gap.

5.7 Hardening vs. reverse engineering attacks and tools

Attacks perhaps are the least studied (and published) part of the integrity protection schemes. This is due to two main factors: **a)** plenty of security measures in industrial protection schemes such as Themida and VMProtect are not publicly disclosed, which in turn, has left researcher without enough knowledge about their security. **b)** Breaking schemes is unethical as it opens the gate for the attackers to compromise system's security. For example, the CIA Vault7 windows file sharing exploit has led to WannaCry ransomware which infected 213,000 windows machines in 112 countries [7].

Due to the lack of comprehensive research on attacks on integrity protection schemes, evaluating their security against various attacks is impossible. Therefore, as an initial step towards analyzing the security of these schemes, we made assumptions regarding the resilience of protection schemes against different attacks. In the following we state our assumptions:

1. **Disassembler:** code concealment and layered interpretation hardening measures hinder disassembly. Code concealment techniques commonly affect the correctness of static disassemblers [57].

2. **Debugger:** in a strict sense only schemes that directly utilize anti-debugger measures impede debug based attacks. These technique are, however, ad-hoc [2]. In this work, we consider hardening measures that impose some difficulties on the ability of debugging programs as counter debug measures. Cyclic checks, layered protection and mutation are the three hardening measures that we believe impede debug based attacks. Cyclic checks confuse attackers and exhaust their resources. Layered protection complicates the program execution flow and thus has an impact on the effectiveness of debugging. Mutation-based techniques renders debug knowledge useless after each mutation. Furthermore, SGX protected application cannot be debugged in production.

Table 9: MATE tools resilience in the reviewed schemes.

Disassembler	Debugger	Tracer
[6, 10, 8, 60, 74, 88]	[8, 11, 13, 19, 21, 31, 33, 37, 36, 40, 46, 54, 60, 67, 91]	[11, 23, 36, 41, 56, 60, 91]

3. **Tracer:** enable attackers to monitor what instructions are actually being executed by the program. With the help of this tool, attackers could effectively bypass code concealment and virtualization as program instructions have to be translated and eventually executed. This is the point that a tracer can dump the plain instructions. Nevertheless, frequent program mutation renders captured data useless. Furthermore, code clones are normally harder to capture using a tracer and thus it hinders tracer based attacks. *Intel SGX isolation*, by design, resist against tracers.

4. **Emulator:** enables adversaries to analyze protected programs in a revertible and side-effect free environment. In effect, this defeats response mechanism and gives unlimited attempts to attackers. Beside anti-emulation measures, which are again ad-hoc, only hardware based security appears to impede emulators. However, emulation on its own does not break an scheme, as attackers have to utilize other tools to detect and ultimately defeat the protection. Simply put, emulation acts as a facilitator for attacks. Therefore, we exclude emulator resilience from our analysis.

With the given assumption we have classified the reviewed literature based on their hardening measures. Table 9 represents the mapping between different schemes and their resilience to MATE Tools.

5.8 Resilience against known attacks

As mentioned earlier, the security of integrity protection schemes has not been thoroughly evaluated, due to the obscurity and lack of access to the resources. This has led to very few attempts on breaking such schemes.

To the best of our knowledge, three generic attacks on integrity protection schemes were published, two technical attacks and one conceptual work for manifesting attacks. In the following we discuss these attacks and map them to the hardening measures that could potentially address them.

1. **Pattern matching:** There are no paper on pattern matching attacks on different protections schemes. Additionally, most of the papers mention obfuscation as the golden hammer against pattern matching and entropy measurements. Unfortunately, we cannot evaluate resilience of the reviewed schemes against pattern matching as their source code is not available. Therefore, we do not consider this attack in the mapping process.

2. **Memory split:** Wurster et al. in [89] proposed an attack to defeat self-hashing protection schemes by redirecting self-reads to an untampered version of the program loaded in a different memory address. For this purpose, they modified the kernel to establish two distinct memory pipelines for self-read and instruction fetch (execution). In this setting, the execution pipeline can readily be tampered with by attackers and self-reads remain unaware of such modifications.

   To address this generic attack, Giffin et al. in [38] proposed self-modifying code defense mechanism. The idea is to add a token to the program code at runtime and verify it

accordingly. Dynamically generated tokens can detect redirection of self-memory reads in the protected processes.

Self-modifying code, despite its effectiveness in thwarting memory split attacks, opens the door for other integrity violating attacks. The reason being that enabling self-modification requires flagging writable memory pages as executable pages in the program. This raises the concerns about code injection attacks [4], in which maliciously prepared memory blocks could potentially get executed.

This attack only applies to introspection based techniques, i.e. techniques in which a program's code blocks have to be read at runtime. Thus, state inspection based techniques are resilience against memory split attack. In addition to that, self mutating protection schemes express the same effect as the defense that was proposed by [4]. All in all, our survey shows state inspection based and mutation based techniques mitigate the memory split attack.

3. **Taint analysis to detect self-checksumming:** Qiu et al. in [75] proposed a technique to detect self-checksumming guards in programs. Their attack relies on the fact that these class of protections forces a protected program to read its own memory at runtime. This obligation enabled them to utilize dynamic information flows in detection of self checksumming guards and check conditions (which trigger the response mechanism). Technically speaking, their approach captures a trace of the program (using Intel Pin) and then searches for a value X which is tainted by self memory values (backward taint analysis) and then detects those Xs that are used in a condition (forward taint analysis).

   This attack appears to be the ultimate attack on self-checking protection schemes as it can in theory detect all checks in a given program. However, tracing medium to large sized programs will generate massive trace logs that have to be analyzed. [8] reported analyzing a small trace (captured in 10 minutes) of Google Chromium using the tainting technique requires a long time in which only 1% of the traces were analyzed in a day.

   Since this attack relies on a Tracer tool (Intel Pin), we can mark schemes that resist against tracer as resilient to the attack presented in [75]. Needless to say that larger programs, due to the complexity of the backward and forward taint analysis, in general cannot be targeted by this attack in a reasonable time.

4. **Graph based analysis:** Dedic et al. in [29] indicate that flow graph analysis such as pattern matching and connectivity analysis (given that checker nodes are weakly connected to other graph nodes in a program) can help defeat the existing protection schemes. They developed a graph based game to formally present these attacks.

   Against these attacks, they proposed three design principles, viz. *strongly connected checks*, *distributed checks* and *threshold detection schemes*, that can significantly harden the identification and disabling of a tamper protection technique. Strongly connected checks requires checkers to be strongly connected to other nodes of the program. Distributed checks refers to a network of checkers which an attacker has to disable them all in order to successfully tamper with an application. Threshold detection schemes suggests to call a response function only when a k number of checks failed. This prevents attackers from sequentially detecting checkers in the program.

   This work can be seen as a conceptual work with no actual implementation of the defenses or attacks. It would be interesting to employ the suggested measures and carry out the

[1] Intel pin is a dynamic program instrumentation tool

Table 10: Resilient schemes against the two known technical attacks.

Taint based attack	Memory split attack
[11, 23, 36, 41, 56, 60, 91]	[1, 10, 12, 13, 16, 17, 20, 23, 34, 36, 40, 41, 42, 43, 44, 49, 52, 60, 62, 63, 67, 69, 70, 79, 81, 83, 85]

attack to verify their claim. None of the reviewed schemes appears to utilize all the defenses together. Therefore, it might be the case that in theory all of the schemes are defeated by the graph based attack. Since this attack has not been implemented in practice, we rather leave it for further evaluations.

In Table 10 we map the schemes that resist against the two practical attacks.

6 Related Work

The closest and yet distant research to our work is ASPIRE (`www.aspire-fp7.eu`) project. It is a project funded by European Union for software protection. ASPIRE is designed to facilitate software protection by introducing a reference architecture for protection schemes. Conformance to this architecture enables practitioners to compose a chain of protections simultaneously, which offers more resilience [84]. Data hiding, code hiding, tamperproofing, remote attestation and renewability are the core protection principles that are addressed in ASPIRE.

ASPIRE introduces a security policy language expressed by annotations. To protect a program using this framework, end-users have to annotate programs with their desired security properties. A compiler tool chain is designed which analyzes these annotations and carries out necessary protection steps. To do so, the tool chain comes with three components: a source code transformation engine powered by TXL [28], a set of routines to link external protection schemes compatible with standard compilers (LLVM and gcc), and a link-time binary rewriting infrastructure for post-compilation protections powered by Diablo (`http://diablo.elis.ugent.be/`). All these enable rapid development of protection tools which could be used by different programs in a customizable and yet composable manner.

The ASPIRE framework focuses on a technical architecture for design and employment of a wide range of protection mechanisms (not only integrity protection). However, both the programs to protect and the protection schemes have to conform to the requirements of the tool chain. Specifically, the source code (in C/C++ language) of programs is mandatory. Furthermore, constraints of each scheme need to be specified in its manifest. The framework uses these manifests to report potential conflicts to users based on which they can identify and subsequently single out conflicting measures.

The ASPIRE project has not provided a taxonomy of integrity protection schemes and does not include comparison of different schemes. In fact, the outcome of this work can contribute in extending the ASPIRE architecture to accommodate a wider range of integrity protection schemes, namely by means of introducing new requirements for the annotation language (such as desired resilience against certain attacks), and extensions for the support of further protection schemes (such as hypervisor based and hardware assisted ones).

To the best of our knowledge, our work is the first taxonomy of integrity protection schemes. Therefore, reviewing existing taxonomies was not an option for us. Instead, we reviewed scheme classifications and surveys on integrity protection. In the following we report on the reviewed publications.

6.1 Existing Classifications

Mavrogiannopoulos et al. in [64] propose a taxonomy for self-modifying obfuscation techniques, which essentially is based on the concept of mutation.

Collberg et al. in [26] classify integrity protection techniques into four main categories: *self-checking*, *self-modifying*, *layered interpretation* and *remote tamper-proofing*. In a similar effort, Bryant et al. [15] classify tamper resistant systems into 6 categories, viz. hardware assisted, encryption, obfuscation, watermarking, fingerprinting and guarding. Considering the fact that watermarking on its own does not contribute to integrity protection, we exclude it from our analysis. Guarding also fits the self-checking category, thus it can be omitted. Likewise encryption is a substance of self-modifying primitive. After we resolved the conflicts, we study the structure of the five remaining categories of integrity protection techniques as follows.

Self-checking. This refers to a technique in which self-unaware programs are transformed into somewhat self-conscious equivalent versions. The transformation is done via equipping programs with a set of monitoring probes (checkers/testers), that monitor *some authentic features* of the target program, along with a group of assertions, who compare probe results to the corresponding *known expected values* and take *appropriate actions* against tampering attacks.

Depending on which features of a program are being monitored, self-checking itself has two subcategories: *Introspection* and *State Inspection*. The former exercises the shape of a program, for instance code blocks, whereas the latter is after monitoring the program's execution effects, for example the sum of a program's constant variables. Among the two, state inspection is more appealing because, unlike the introspection, it reflects the actual execution of a program, and thus it is harder to counterfeit. However, monitoring dynamic properties of a program is more difficult compared to the static code shape verification.

Self-modifying. Turning a program into a running target by mutating it at runtime is the idea behind self-modifying techniques. The majority of schemes that are based on this technique use encryption as the mean to mutate and ultimately to protect a program. However, there are some schemes that use instruction re-ordering, instead of encryption, to mutate the program.

Mavrogiannopoulos et al. in [64] define four main criteria in their proposed taxonomy for self-modifying obfuscation techniques: *concealment* (the size of program slices that are being modified at each mutation interval), *encoding* (the technique that is used to mutate the code, e.g. instruction reordering or encryption), *visibility* (whether the code is entirely or partially protected) and *exposure* (whether the actual code is permanently or temporarily obtainable at runtime). Nevertheless, in a more recent survey, virtualization was also proposed as a relevant technique in [90], but we rather consider virtualization as a subcategory of layered interpretation techniques.

Sasirekha et al. in [77] review 11 software protection techniques. Their research confirms the trade off between and security and performance for all the reviewed schemes based on which they suggested compiler-level protection as possible direction for further research. However, their work lacks any classification of techniques.

Layered interpretation. In this approach a program's instructions are replaced with a set of seemingly random byte codes. These codes can only be executed with a program specific emulator that is normally shipped with the program itself. Tampering with the program byte code may result in unrecognized byte code in the emulator and may eventually causes failures. However, some techniques (for instance [54]) perform integrity verifications at the emulator prior to executions. In general, layered interpretation techniques add resilience against static program

analysis attacks. To the best of our knowledge, there is also no published taxonomy on layered interpretation protection methods.

Remote tamper-proofing. Protection schemes that enables a software systems to be verified externally fall under this category. In an abstract sense, remote tamper proofing has very much in common with *remote attestation* concept. Since remote attestation is a vast field of research that deserve a taxonomy on its own, its classification falls out of the scope of this paper.

6.2 Comparison of schemes and surveys

Dedic et al. in [29] define *distributed check* and *threshold based detection* as significant resilience improver factors in integrity protection schemes. They use a graph based game and theoretically prove the enhanced complexity of the scheme with the aforementioned factors. From this work two interesting features are extracted: *detection* and *response algorithms*.

In an effort for security classification and unification of protection notation, Collberg et al. after defining a notation, use natural science and human history to derive a set of 11 unique defense primitives in [25]. These primitives state the defense concepts rather than security guarantees.

In an effort for MATE attack classification, Akhunzada et al. in [3] study different types and motives behind these attacks. As another outcome of their work, they stressed that performance factor is one of the main pain points of protection techniques. However, it appears that no study tackled this matter and there is a lack of a benchmark for protection schemes [3]. The focus of their work is not the techniques-to-protect themselves, but rather the attackers. Consequently, the security guarantees are not studied.

The main gap here is lack of a holistic view software protection process. The process itself is not well defined, business requirements are not elicited, constraints are not studied, and more importantly, security guarantees that different schemes offer are not studied.

7 Future of integrity protection

Integrity protection is a crucial subject in cyber security. We believe this subject will attract more attention as the trend of digitalization expands into more domains of our daily life. As a matter of fact, the Internet of Things (IoT) is one of the major domains which requires immediate action w.r.t. integrity protection. The reason is the exposure of IoT devices to MATE attackers.

Integrity protection will be impacted by three dimensions: **a)** trusted hardware, **c)** computer networks and **(c)** complexity of software systems. As a consequence of advances in trusted hardware, software based integrity protection will perhaps be slowly replaced by more hardware based or hybrid methods. Simply put, advancements in secure and yet cheap hardware equipments will be a major influence on the transition from software based to hardware aided integrity protection. However, this transition will require a new adaptation of the current trust model utilized by trusted hardware.

Network advances can enable a far more flexible software systems with respect to connectivity requirements. Currently, enforcing constant connections between a remote trusted party and a client system is undesirable in many contexts, e.g. the automotive industry. Lower latency and higher availability in communication protocols can foster a new model of integrity protection that far more relies on remote parties.

Beside hardware and network improvements, software systems are getting more complex as a natural response to complex problems. This negatively impacts integrity protection, as it

becomes more challenging for users to apply protections. Therefore, integrity protection requires a solution to reduce the complexity of protection.

In the following we have a closer look at the aforementioned influencing dimensions.

7.1 Trusted hardware and collective trust model

Advances in hardware aided integrity could offer better program isolation and hence better integrity. Moving towards future, secure hardware should become more and more common in new computers. Dedicated hardware can improve integrity protection. However, trusted hardware is and will not be silver bullet. Side channels remain a problem in such systems. Recent attack proposed in [78] on SGX protected program enabled attackers to extract RSA keys from protected (fully isolated) enclaves. This proves that integrity protection against MATE attacks remains a cat-mouse-game.

Hardware security can potentially improve integrity protection in software systems. Nevertheless, this is a double edged sword, as malware can also benefit from such technologies to hide their malicious behavior from anti viruses or malware analysts. To cope with this problem, Intel SGX requires protected programs (enclaves) to be signed by Intel in production machines. This implies trusting Intel with holding signing keys. Intel argues this is necessary for two reasons. First, they can sell licenses to use their security technology, without purchasing a license they will not sign anything. Second, they can control who can use their security system, in an effort to ensure malware producers are unable to misuse Intel's ecosystem.

The current model, however, clearly resembles a single point of failure. If the current trend of block chain and collective trust model persists, it is likely that Intel will not remain the only player. Most probably a block chain alike system will replace the current model. One naive realization would be to have a group of semi trusted entities (with conflicting interests) to sign protected applications. Mapping this to SGX means that n entities will sign enclaves in addition to Intel. At the user end, the protected program will be executed, if and only if k signatures match, where $0 < k \leq n$.

A more interesting realization would be to enable users to report any misbehavior of the protected program by broadcasting a message to a set of disjoint logging entities. Then a set of third party verifiers (similar to miners in block chain) can verify these broadcast list and subsequently warn new users about the malicious protected packages. The main benefit of this approach is that there is no single point of failure.

7.2 Goal based protection

Currently, most protection schemes require the user to specify sensitive regions/data in the program to be protected. The protection scheme then runs a transformation in order to safeguard the specified items. For instance, Intel SGX requires users to implement their application in two regions: trusted and untrusted. The trusted region will access sensitive data and hence shall be protected by secure enclaves. Software advances along integrity protecting hardwares can enable a smooth transition from manual software transformation to automatic or at least guided transformations. One possibility is to have users specify high level integrity goals, instead of marking code regions, then some static analyzers scan the program and automatically mark regions of the program to be protected. This not only reduces the effort on the user end, but also improves the precision of the protection.

Once these sensitive regions are identified, then hardware integrity could be utilized to protect them. For instance, sensitive regions could be automatically moved into SGX trusted zone and necessary signature changes could be applied to minimize the user intervene.

7.3 Tactile Internet

Real-time constraints are one of the main reasons to deliver applications to untrusted clients. Recent advancement in cloud computing have greatly extended this model by migrating application to client-server paradigm. Servers implement functionality of the system and deliver results to clients as per their requests. Today, we have fully functional word processing applications hosted on the cloud and enabled users to interact with them via their browser. In this setting, software vendors need not to worry about license check circumvention as end users do not have access to the program binaries nor execution environment. A secure authentication mechanism in effect handles license verifications.

However, given the current latency and availability of Internet networks, shifting to enforcing clients to stay in direct and constant connections with servers imposes substantial overheads. In a system with real-time constraint, e.g. the automotive industry, such delays are unacceptable. Thus, a thick layer of software systems has to reside on the client side to deliver the functionality within an acceptable time frame.

Advances in tactile Internet, which guarantees insignificant latency and high availability [61], may render the need for distributing softwares obsolete, by enabling the execution of softwares on trusted servers and communicating results to endusers. The tactile Internet can change the current model, by moving the control system away into a trusted secure environment. Moving the sensitive code into trusted servers does not fully mitigate integrity attacks, however. For instance, attackers may manage to intercept and forge control commands send to a connected car. Another attack could be to manipulate internal equipments to behave faulty. Nevertheless, this shift reduces the attack surface and effectively protects proprietary data.

All in all, integrity protection will move towards a more hardware protected system with a major dependency on network reachability in realtime. Moreover, user's role in the protection process will be as much as defining high-level integrity goals after. The rest of the protection will be handled by the system automatically.

8 Conclusions

MATE attackers have successfully executed serious integrity attacks far beyond disabling license checks in software system, to the extent that the safety and security of users is at stake.

Software integrity protection offers a wide range of techniques to mitigate a variety of MATE attacks. Despite the existence of different protection schemes to mitigate certain attacks on different system assets, there is no comprehensive study that compares advantages and disadvantages of these schemes. No holistic study was done to measure completeness and more importantly effectiveness of such schemes in different industrial contexts.

In this work, we therefore presented a taxonomy for software integrity protection which is based on the protection process. This process starts by identifying integrity assets in the system along with their exposure in different asset representations throughout the program lifecycle. These steps are captured in the system view, which is the first dimension of our taxonomy. In the next dimension, the attack view, potential threats to assets' integrity are analyzed and desirable protections against particular attacks and tools are singled out. Then, the defense dimension sheds light on different techniques to satisfy the desired protection level for assets at different representations. This dimension also serves as a classification for integrity protection techniques by introducing 8 unique criteria for comparison, viz. protection level, trust anchor, overhead, monitor, check, response, transformation and hardening. To support the understandability of the taxonomy we used a fictional DRM case study and mapped it to the taxonomy elements.

We also evaluated our taxonomy by mapping over 49 research articles in the field of integrity protection. From the mapped articles we correlated different elements in the taxonomy to address practical concerns with protection schemes, and to identify research gaps.

In the correlation of assets and representations on which protections are applied we identified relevant schemes for protecting different integrity assets (behavior, data and data and behavior). This facilitates the scheme selection based on the assets and representations at the user end. A major gap was identified in protecting integrity of distributed software systems.

In MATE vs. representation correlation we have shown different schemes that mitigate different MATE attacks, viz. binary, control flow, process memory and runtime data, at different asset representations.

In the correlation of transformation lifecycle and protected representations, we classified schemes based on their representation-to-protect and their applicability on different program lifecycles. Our results indicates a gap in compiler based, load and run time protection schemes.

To study the impact of the transformation lifecycle on the protection overhead, we correlated transformation lifecycles and scheme overheads. Despite the general belief that compiler based protection should perform better, our results suggest that post-compilation transformations perform quite optimal. Nevertheless, we were unable to verify the performance of compiler based protections due to lack of benchmark results. We believe the lack of reliable benchmark data as well as a benchmarking infrastructure is a significant gap in software integrity protection. This is due to the fact that only implementation of a few protection schemes were made public, for instance in self-checking schemes only [8] has published their source code. Thus, evaluating their performance, without re-implementing them, is infeasible.

The correlation between checking mechanism in different protection schemes suggests that checksumming is the most used technique for integrity verification. Other checking measures are equally unpopular. The correlation of representation and protection level shows a gap in hypervisor based protections.

Another major gap in the literature is to analyze resilience of protection schemes against attacks. Based on our classification data we evaluated the resilience of reviewed schemes against MATE tools and known attacks on integrity protections. In the resilience against MATE tools analysis, we proposed a set of assumptions for declaring whether a scheme resists again a certain MATE tool or not. The assumptions are based on the hardening techniques that are used by protection schemes. These assumptions were then used to classify schemes based on resilience against MATE tools.

In the last correlation we analyzed the resilience of reviewed schemes against known attacks on protection schemes. Our Survey indicated that there is a gap in studying the resilience of protection schemes against pattern matching attacks. This is again due to a lack of any benchmark of different techniques whatsoever. Regarding memory split and taint based attacks, we first identified hardening measures that impedes such attacks. Subsequently, we marked schemes as resilient to memory split and taint based attacks according to their hardening measures.

All in all, a big gap in integrity protection research is the lack of a benchmark of performance and security guarantees, both of which require a dataset of integrity protection mechanisms.

Limitation: The proposed taxonomy only serves as a help for practitioners. We do not claim that it is exhaustive nor complete w.r.t. industry standard protection processes. A good direction for future work is to map the taxonomy on standard processes such as Microsoft Secure Development lifecycle [66].

Our taxonomy puts more emphasis on the defense mechanisms, as the number of published attacks on integrity protections is incomparably fewer than published defenses. Mapping concrete attacks will be a good extension to our taxonomy.

Although we included some research papers on infrastructural security, the main focus of our analysis was application level integrity protection. In reality, there are plenty of ways in the infrastructure that adversaries can potentially exploit. Such exploits could lead to circumvention of defense mechanisms and potentially violation of the application integrity. For instance, one major attack that we did not address in this work is call interception (also known as call hooking), which enables attackers to run malicious code by swapping the address of external calls. Another interesting direction is to map infrastructural integrity protection measures along with known attacks on them to the taxonomy.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] B. Abrath, B. Coppens, S. Volckaert, J. Wijnant, and B. De Sutter. Tightly-coupled self-debugging software protection. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, SSPREW '16, pages 7:1–7:10, New York, NY, USA, 2016. ACM.

[3] A. Akhunzada, M. Sookhak, N. B. Anuar, A. Gani, E. Ahmed, M. Shiraz, S. Furnell, A. Hayat, and M. K. Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications*, 48:44–57, 2015.

[4] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277. IEEE, 2008.

[5] B. Anckaert, M. Jakubowski, and R. Venkatesan. Proteus: virtualization for diversified tamper-resistance. In *Proceedings of the ACM workshop on Digital rights management*, pages 47–58. ACM, 2006.

[6] D. Aucsmith. Tamper resistant software: An implementation. *Proceedings of the First International Workshop on Information Hiding*, pages 317–333, 1996.

[7] Avast Lab. Wannacry ransomware, 2017.

[8] S. Banescu, M. Ahmadvand, A. Pretschner, R. Shield, and C. Hamilton. Detecting patching of executables without system calls. In *Proceedings of the Conference on Data and Application Security and Privacy*, 2017.

[9] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200. ACM, 2016.

[10] S. Banescu, A. Pretschner, D. Battré, S. Cazzulani, R. Shield, and G. Thompson. Software-based protection against changeware. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 231–242. ACM, 2015.

[11] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[12] B. Blietz and A. Tyagi. Software tamper resistance through dynamic program monitoring. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3919 LNCS:146–163, 2006.

[13] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: tiny trust anchor for tiny devices. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[14] S. Brockmans, R. Volz, A. Eberhart, and P. Löffler. Visual modeling of owl dl ontologies using uml. In *International Semantic Web Conference*, volume 3298, pages 198–213. Springer, 2004.

[15] E. D. Bryant, M. J. Atallah, and M. R. Stytz. A survey of anti-tamper technologies. *CrossTalk*, pages 12–16, November 2004.

[16] M. Carbone, W. Cui, M. Peinado, L. Lu, and W. Lee. Mapping Kernel Objects to Enable Systematic Integrity Checking. *Analysis*, pages 555–565, 2009.

[17] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[18] L. Catuogno and I. Visconti. A format-independent architecture for run-time integrity checking of executable code. In *International Conference on Security in Communication Networks*, pages 219–233. Springer, 2002.

[19] H. Chang and M. J. Atallah. Protecting software code by guards. In *ACM Workshop on Digital Rights Management*, pages 160–175. Springer, 2001.

[20] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *International Workshop on Information Hiding*, pages 400–414. Springer, 2002.

[21] M. Christodorescu, R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni. Cloud security is not (just) virtualization security: a short paper. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 97–102. ACM, 2009.

[22] C. Collberg. Defending against remote man-at-the-end attacks, 2017.

[23] C. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 319–328. ACM, 2012.

[24] C. Collberg and J. Nagra. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection.* Addison-Wesley Professional, 2009.

[25] C. Collberg, J. Nagra, and F.-Y. Wang. Surreptitious software: Models from biology and history. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 1–21. Springer, 2007.

[26] C. S. Collberg, I. C. Society, C. Thomborson, and S. Member. Obfuscation Tools for Software Protection. 28(8):735–746, 2002.

[27] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering*, 28(8):735–746, 2002.

[28] J. R. Cordy. Txl-a language for programming language tools and applications. *Electronic notes in theoretical computer science*, 110:3–31, 2004.

[29] N. Dedić, M. Jakubowski, and R. Venkatesan. A graph game model for software tamper protection. In *International Workshop on Information Hiding*, pages 80–95. Springer, 2007.

[30] Y. Deswarte, J.-J. Quisquater, and A. Saïdane. Remote integrity checking. In *Integrity and internal control in information systems VI*, pages 1–11. Springer, 2004.

[31] P. Dewan, D. Durham, H. Khosravi, M. Long, and G. Nagabhushan. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proceedings of the 2008 Spring simulation multiconference*, pages 828–835. Society for Computer Simulation International, 2008.

[32] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[33] J. Gan, R. Kok, P. Kohli, Y. Ding, and B. Mah. Using virtual machine protections to enhance whitebox cryptography. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 17–23, May 2015.

[34] Z. Gao, N. Desalvo, P. D. Khoa, S. H. Kim, L. Xu, W. W. Ro, R. M. Verma, and W. Shi. Integrity protection for big data processing with dynamic redundancy computation. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pages 159–160, July 2015.

[35] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 193–206. ACM, 2003.

[36] S. Ghosh, J. Hiser, and J. W. Davidson. Software protection for dynamically-generated code. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 1. ACM, 2013.

[37] S. Ghosh, J. D. Hiser, and J. W. Davidson. A secure and robust approach to software tamper resistance. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6387 LNCS:33–47, 2010.

[38] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 10–pp. IEEE, 2005.

[39] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance. *Security and Privacy in Digital Rights Management*, pages 141–159, 2002.

[40] A. Ibrahim and S. Banescu. Stins4cs: A state inspection tool for c. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 61–71. ACM, 2016.

[41] M. Jacob, M. H. Jakubowski, and R. Venkatesan. Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In *Proceedings of the 9th workshop on Multimedia & security*, pages 129–140. ACM, 2007.

[42] M. Jakobsson and K.-A. Johansson. Retroactive detection of malware with applications to mobile platforms. In *Proceedings of the 5th USENIX Conference on Hot Topics in Security*, HotSec'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

[43] M. Jakobsson and K.-A. Johansson. Practical and secure software-based attestation. In *Lightweight Security & Privacy: Devices, Protocols and Applications (LightSec), 2011 Workshop on*, pages 1–9. IEEE, 2011.

[44] H. Jin and J. Lotspiech. Forensic analysis for tamper resistant software. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 133–142. IEEE, 2003.

[45] P. Johnson, A. Vernotte, M. Ekstedt, and R. Lagerström. pwnpr3d: an attack-graph-driven probabilistic threat-modeling approach. In *Availability, Reliability and Security (ARES), 2016 11th International Conference on*, pages 278–283. IEEE, 2016.

[46] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-llvm: software protection for the masses. In *Proceedings of the 1st International Workshop on Software Protection*, pages 3–9. IEEE Press, 2015.

[47] T. Kanstrén, S. Lehtonen, R. Savola, H. Kukkohovi, and K. Hätönen. Architecture for high confidence cloud security monitoring. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 195–200. IEEE, 2015.

[48] A. K. Kanuparthi, M. Zahran, and R. Karri. Architecture support for dynamic integrity checking. *IEEE Transactions on Information Forensics and Security*, 7(1):321–332, 2012.

[49] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-end integrity protection for web applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2016.

[50] S. Karnouskos. Stuxnet worm impact on industrial cyber-physical system security. In *IECON 2011-37th Annual Conference on IEEE Industrial Electronics Society*, pages 4490–4494. IEEE, 2011.

[51] Kaspersky Lab. Projectsauron: top level cyber-espionage platform covertly extracts encrypted government comms, 2017.

[52] C. Kil. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. *IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 115–124, 2009.

[53] G. H. Kim and E. H. Spafford. Experiences with tripwire: Using integrity checkers for intrusion detection. 1994.

[54] W. B. Kimball and R. O. Baldwin. Emulation-based software protection, Oct. 9 2012. US Patent 8,285,987.

[55] B. Kordy, P. Kordy, S. Mauw, and P. Schweitzer. Adtool: security analysis with attack–defense trees. In *International Conference on Quantitative Evaluation of Systems*, pages 173–176. Springer, 2013.

[56] A. Kulkarni and R. Metta. A new code obfuscation scheme for software protection. In *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*, pages 409–414. IEEE, 2014.

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

[58] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu. Repackage-proofing android apps. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pages 550–561. IEEE, 2016.

[59] S. Luo and P. Yan. Fake apps: Feigning legitimacy. Technical report, Trend Micro, 2014.

[60] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. In *International Workshop on Information Security Applications*, pages 194–206. Springer, 2005.

[61] M. Maier, M. Chowdhury, B. P. Rimal, and D. P. Van. The tactile internet: vision, recent progress, and open challenges. *IEEE Communications Magazine*, 54(5):138–145, 2016.

[62] C. Malone, M. Zahran, and R. Karri. Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs. *Proceedings of the sixth ACM workshop on Scalable trusted computing - STC '11*, page 71, 2011.

[63] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: Tamper-proof code execution on legacy systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6201 LNCS:21–40, 2010.

[64] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[65] R. Merkle. A digital signature based on a conventional encryption function. In *Advances in CryptologyâĂŤCRYPTOâĂŹ87*, pages 369–378. Springer, 2006.

[66] Microsoft. Simplified implementation of the microsoft sdl, 2017.

[67] B. Morgan, E. Alata, V. Nicomette, M. Kaâniche, and G. Averlant. Design and implementation of a hardware assisted security architecture for software integrity monitoring. In *Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium on*, pages 189–198. IEEE, 2015.

[68] R. Neisse, D. Holling, and A. Pretschner. Implementing trust in cloud infrastructures. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 524–533. IEEE Computer Society, 2011.

[69] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[70] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, 2013.

[71] S. Park, J. N. Yoon, C. Kang, K. H. Kim, and T. Han. Tgvisor: A tiny hypervisor-based trusted geolocation framework for mobile cloud clients. In *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2015 3rd IEEE International Conference on*, pages 99–108. IEEE, 2015.

[72] U. Piazzalunga, P. Salvaneschi, F. Balducci, P. Jacomuzzi, and C. Moroncelli. Security strength measurement for dongle-protected software. *IEEE Security Privacy*, 5(6):32–40, Nov 2007.

[73] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over gf (p) and its cryptographic significance (corresp.). *IEEE Transactions on information Theory*, 24(1):106–110, 1978.

[74] M. Protsenko, S. Kreuter, and T. MÃijller. Dynamic self-protection and tamperproofing for android apps using native code. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 129–138, Aug 2015.

[75] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su. Identifying and Understanding Self-Checksumming Defenses in Software. pages 207–218, 2015.

[76] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[77] N. Sasirekha and M. O. Hemalatha. A Survey on Software Protection Techniques. 12(1), 2012.

[78] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using sgx to conceal cache attacks. *arXiv preprint arXiv:1702.08719*, 2017.

[79] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. *ACM SIGOPS Operating Systems Review*, 2005.

[80] H. Shacham, E. Buchanan, R. Roemer, and S. Savage. Return-oriented programming: Exploits without code injection. *Black Hat USA Briefings (August 2008)*, 2008.

[81] D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security*, 3(1):51–62, 2000.

[82] D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):51–62, 2000.

[83] Y. Sun, S. Nanda, and T. Jaeger. Security-as-a-service for microservices-based cloud applications. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 50–57. IEEE, 2015.

[84] B. D. Sutter, P. Falcarin, B. Wyseur, C. Basile, M. Ceccato, J. DAnnoville, and M. Zunke. A reference architecture for software protection. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 291–294, April 2016.

[85] F. A. Teixeira, G. V. Machado, F. M. Pereira, H. C. Wong, J. Nogueira, and L. B. Oliveira. Siot: securing the internet of things through distributed system analysis. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 310–321. ACM, 2015.

[86] N. Tillmann and P. de Halleux. Pex - white box test generation for .net. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966, page 134âĂŞ153, Prato, Italy, April 2008. Springer Verlag.

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

[88] P. Wang, S.-k. Kang, and K. Kim. Tamper Resistant Software Through Dynamic Integrity Checking. *Proc. Symp. on Cyptography and Information Security (SCIS 05)*, 2005.

[89] G. Wurster, P. C. Van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. *Proceedings - IEEE Symposium on Security and Privacy*, pages 127–135, 2005.

[90] M. Xianya, Z. Yi, W. Baosheng, and T. Yong. A survey of software protection methods based on self-modifying code. In *Computational Intelligence and Communication Networks (CICN), 2015 International Conference on*, pages 589–593. IEEE, 2015.

[91] F. Yao, R. Sprabery, and R. H. Campbell. Cryptvmi: a flexible and encrypted virtual machine introspection system in the cloud. In *Proceedings of the 2nd international workshop on Security in cloud computing*, pages 11–18. ACM, 2014.