View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by PORTO@iris (Publications Open Repository TOrino - Politecnico di Torino)

POLITECNICO DI TORINO
Repository ISTITUZIONALE

Software Attestation with Static and Dynamic Techniques

Original
Software Attestation with Static and Dynamic Techniques / Viticchie', Alessio. - (2019 Jul 17), pp. 1-114.

Availability:
This version is available at: 11583/2749160 since: 2019-09-02T09:56:42Z

Publisher:
Politecnico di Torino

Published
DOI:

Terms of use:
openAccess

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

04 August 2020

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (31st cycle)

Software Attestation with Static and Dynamic Techniques

Alessio Viticchié

* * * * * *

Supervisors
Prof. Antonio Lioy, Supervisor
Cataldo Basile, Ph.D., Co-supervisor

Doctoral Examination Committee:
Prof. Bjorn De Sutter, Referee, Ghent University
Prof. Stefano Paraboschi, Referee, Università degli Studi di Bergamo
Bart Coppens, Ph.D., Ghent University
Prof. Claudia Raibulet, Università degli Studi di Milano-Bicocca
Prof. Riccardo Sisto, Politecnico di Torino

Politecnico di Torino
17 July 2019

This thesis is licensed under a Creative Commons License, Attribution - Noncommercial - NoDerivative Works 4.0 International: see www.creativecommons.org. The text may be reproduced for non-commercial purposes, provided that credit is given to the original author.

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . .

Alessio Viticchié

Turin, 17 July 2019

Abstract

The spread of software tools in every field of modern daily life is forcing the need for effective software integrity protection techniques. This kind of protection is becoming necessary in order to avoid economic threats for producers and security threats for users. Software protection is particularly needed when applications run in untrusted environments (e.g. mobile devices or personal computers). Indeed, it is non-trivial to ensure application security when the attacker controls the entire execution environment. This scenario has recently defined the new class of attacks: Man-At-The-End (MATE). The application of protection techniques against these attacks could be challenging for developers because they often have limited knowledge in software protection. Furthermore, protections techniques presented in literature have critical limitations when applied in MATE scenarios (e.g. not effective enough, not practical or too complex to be applied). As a final issue, modern systems are very diverse, and their configurations are highly varied. Consequently, software protection techniques must be independent of any underlying configuration in order to suit modern systems. Hence, it is worth to investigate new software protection techniques and automatic methods to apply them to general applications.

This thesis presents *Software Attestation* as a valuable integrity monitoring technique. This work aims at defining the general model of the protection methodology in order to provide a reference architecture and workflow. From the general, abstract model, this work defines the specific versions of the protection as model instantiations that select particular software aspects as assets to protect, on which integrity evidence is computed. Hence, this thesis investigates *Static* and *Dynamic* software attestation as interesting instantiations of the general model. Both these two flavours of attestation are investigated starting by defining the requirements, then proposing possible implementations and finally performing the validation of the requirements and the security analysis. All this discussion aims at providing a comprehensive study about the practical applicability of the protection model. Finally, the dissertation aims at demonstrating that the software attestation model is valid for protection and can be used to achieve more complex and robust protection methodologies by combining it with other methods. Furthermore, this thesis underlines the issues that may come from the actual instantiation of the software attestation model. Hence, the discussion wants to present, once more, that threats generate from practical specifications even when models are theoretically robust.

Puzzled

From the illusion of the inception,
over the lasting duration;
revelation through the best of times

Adorable presence, the fearless men,
lasting stands in the pigpen;
lovely hidden down, the deepest rhymes

Thriving blossoming, deity herald,
hearty hunk of emerald;
envy, not in these hearths, it chimes

Fun and laughs acknowledge the cheerful,
unseen, perfection of God is stealth;
cherish the advice showing own thankful,
kindest protector and plentiful wealth;
shining souls when dark has been painful

Instants, moments, endless memories.

Here and now, mildest spot of serenity,
adorable being of life, of fire and ice;
vicious passion surely enters eternity,
engraves the stone; dazzling star, rise!

Novelty here and oldie there,
ought to hail and revere;
the roses crown in the saint's hand

Graceful silence, God has heard;
infinite appeal, loyal horses made slave.
vice, obsession, etheric land of brave;
elder and latter, present and gone,
now is forever akin the deeds done.

Contents

List of Tables VI

List of Figures VII

1 Background 7
- 1.1 Hardware . . . 8
- 1.2 Software . . . 11
- 1.3 Reaction . . . 15
- 1.4 Open issues . . . 16

2 Architecture 19
- 2.1 Prerequisites . . . 19
- 2.2 Overview . . . 21
- 2.3 The Manager . . . 22
  - 2.3.1 The Master Manager . . . 23
  - 2.3.2 The Slave Manager . . . 24
- 2.4 The Attester . . . 27
- 2.5 The Verifier . . . 29
  - 2.5.1 The Attestation Response Dispatcher . . . 29
  - 2.5.2 The Actual Verifier . . . 29
- 2.6 The Reaction Manager and the Reaction Enforcer . . . 30
- 2.7 The central database . . . 31

3 Detection 37
- 3.1 Preliminary requirements . . . 38
- 3.2 Static software attestation . . . 39
  - 3.2.1 Design . . . 40
  - 3.2.2 Pre-computing attestations: the Extractor . . . 47
  - 3.2.3 Application of static software attestation . . . 47
  - 3.2.4 Security analysis . . . 54
- 3.3 Dynamic software attestation . . . 57
  - 3.3.1 Invariants Monitoring . . . 60

3.3.2 Design 61
3.3.3 Application of Invariants Monitoring 65
3.3.4 Security analysis 69
3.3.5 Discussion 82

4 Reaction 84
4.1 Client-Server Code Splitting 85
4.2 Reactive Attestation 86
4.3 Implementation 88
4.3.1 Annotations processing 88
4.3.2 Static Software Attestation module 89
4.3.3 Client-Server Code-Splitting module 90
4.3.4 Policy engine 92
4.4 Experimental validation 93

5 Conclusions 98

Bibliography 101

List of Tables

2.1 Application table description 34
2.2 Attesters table description 34
2.3 Attest at Start-up table description 34
2.4 Session table description 34
2.5 Request table description 35
2.6 Status table description 35
3.1 Statistical information on the use cases 72
4.1 Effectiveness of Reactive Attestation 95
4.2 Overhead of Reactive Attestation 97

List of Figures

1.1 General Remote Attestation architecture 7
2.1 Software attestation architecture 21
2.2 Scheduling list insertion 25
2.3 Scheduling insertion 26
2.4 Algorithms of software attestation components 36
3.1 Automatic application of static software attestation workflow 52
3.2 Source code that checks a key against a secret 58
3.3 Automatic application of Invariants Monitoring workflow 67
4.1 Example of source code annotated for Reactive Attestation 90
4.2 Pseudo-code of the barrier slicing algorithm 91

Introduction

> Hope proves man deathless.
> It is the struggle of the soul,
> breaking loose from what is perishable,
> and *attesting* her eternity
>
> Henry Melvill

In the last decades, the world underwent, and it is still undergoing, the fastest technological revolution in its history. The digital world has drastically grown in terms of both diffusion and importance in a way that completely changed everyone's daily life. Digital solutions reached more people in the world than any other previous new technology and nearly all the fields of human activities. Digital solutions that involve software tools already aid every daily task. The analysing massive amounts of data from thousands of sensors optimises distribution and reduces leakages in water distribution. The same happens in electricity production and distribution: smart grids intelligently manage power supply to improve the efficiency of the network and reduce wastes. Medical patients profiling and diseases prediction are improved by bioinformatics. Personalised health care was inconceivable in the past, while advanced profiling software tools now make it real. Hybrid and electrical vehicles are now reality both for private and public transportation, software assists the driving tasks and the power consumption optimisations. Factories environments have muted as well: much fewer people and much more machines in plants perform the same old work in a better way. In the *Industry 4.0* model, digital devices and tools lead the production lines and the tasks to perform under the coordination of higher software infrastructures that manage all the aspects of the production such as quality, traceability and auditing, as well as security. Office tasks are all performed using software solutions; documents management, archiving, mailing, stock management, time sheets and agendas are activities that cannot even be imagined without software aid. Mass media, entertainment and marketing have undergone a revolution as well; smartphone apps, websites and smart TVs completely changed the way users access information and enjoy free time.

Consequently, software solutions play a crucial role in this new era of technology. As software applications are deeply involved in many of the new scenarios,

software has become a highly valuable asset also from an economic point of view. An estimation tells that software impacts the global market for about one trillion of euros[1]. Such a massive impact leads to a considerable interest by producers in investing in software development and distribution for any field. The revenue of such an investment is threatened by a set of security issues that come with software distribution. Indeed, the profit of software makers comes from licensed applications, right-reserved contents serving, and advertisements included in free software. Hence, any violation of the delivered applications could result in a severe economic loss for software producers. On the other hand, from the perspective of users, a large set of critical data is managed by software applications. Most of such data are personal data, credentials and sensitive or confidential information. For instance, home banking applications manage bank accounts and users' financial resources, e-commerce applications handle credit cards' numbers and e-mail clients have access to private information and credentials. This kind of data can be leaked by corrupted software to malicious users that could use it to steal money, for identity frauds and information gathering. Hence, corrupted software may cause economic damages, harm the user's business, threaten the user's daily life or family, and even put the user's life at risk. For this reason, techniques and methods to enforce security properties on software application are needed. Integrity is one of the fundamental security properties that have to be guaranteed to avoid risks that may generate from corrupted software.

Modern digital world strongly depends on software and software cracking is a problem that puts at risk both software makers and users. Unfortunately, the software security scenario is made even worse by the surrounding context. The end user has the full control of the delivered software itself and on the environment in which it runs. It means that the final user can exploit sophisticated tools and methodologies to inspect, analyse and modify the software structure or behaviour. Hence, the behaviour of a malicious user (i.e. an attacker) cannot be told from the regular user's one. That is, an attacker who wants to modify an application to alter its original behaviour is inevitably able to do so, this may lead to very stealthy attacks. This context has recently defined a new class of attacks: *Man-At-The-End (MATE)* [3]. MATE attacks describe the cases in which attackers tamper with software applications in contexts where they have full hardware and software privileges, e.g. their computers or mobile devices.

This context introduces the need for software protection techniques able to remotely monitoring the integrity of released software, i.e. the *target*. The literature presented Remote attestation as a valuable response to this need. Remote Attestation is a tampering detection technique that aims at measuring the integrity

[1] https://software.org/wp-content/uploads/2018_EU_Software_Impact_Report_A4.pdf

of software by extracting data from the target in its execution environment and evaluating such data as integrity evidence from a remote node. Traditional trusted computing patterns for remote attestation rely on hardware components, which are assumed to be present in the deploy environments, to establish a root of trust.

Unfortunately, modern distributed computing paradigms involves non-uniform participants; that is, the nature of the network nodes can vary very much. Server-side machines range from classical monolithic hosts running single operating systems to physical hosts running virtual machines, from multiple hosts running several virtual machines to most recent micro-services environments. On the client side, there can be many different software and hardware technologies: traditional personal computers, mobile devices, smart devices, domestic appliances, office devices, and even sensors. Therefore, contemporary networks involve very different devices that may even not be driven by a user, as it happens in modern contexts like Internet-of-Things.

The large variety of deployed technologies makes classical approaches to software security no more applicable. Indeed, it is no longer possible to rely on particular trusted hardware configurations on the client side. Moreover, secure hardware solutions are costly and often absent in most of the deployed devices. Consequently, research resorted to new methods to protect distributed software remotely. Software Attestation is one of the introduced methodologies; it can be considered as the evolution of the classical hardware-based remote attestation. It still aims at detecting tampering by remotely evaluating target integrity, but it does not assume any particular underlying hardware or software component. Hence, different ways to establish the trustworthiness of the client system have been introduced.

Developers represent another critical aspect in software protection: they can be assumed to be neither security-aware nor security experts. Frequently, software developers are skilled enough to realise robust and accurate applications, but they lack the knowledge to implement security remedies in their programs. Hence, software protection techniques have limited value if they are difficult to be manually applied and not supported by automatic procedures.

The so far presented context formerly motivated the work described in the remainder of this document. Then, research effort must be spent in software security to enrich protections knowledge, by both inventing new software protections and continuously improving the existing ones. Moreover, research in software protections has to assess and validate the existing protections to clearly expose the pros and cons of each technique; thus decreeing protection has valuable for real-world applications or not. Indeed, scientific literature presented many protection techniques, they seem valuable and promising, but just a few works continued to study their security properties and effectiveness. In addition, authors often claim their protection techniques to be robust and effective, but no countercheck is provided by other authors. Nonetheless, the software security world must be brought to non-security experts, thus freeing software maker from caring about protection during

the whole development process. Finally, the research about software protection must evolve to represent an opportunity for the security of the whole digital world.

This work identifies *Software Attestation* as a relatively recent protection technique, worthy of being further investigated.

The final goal of this work is to present software attestation as a complete method for software security by demonstrating that: (1) the software attestation model is valid to detect software corruption; (2) software attestation can be seen as a general methodology, which can be easily tailored to monitor different software aspects; (3) the detection abilities of software attestation can assist other techniques in achieving robust protections; (4) software protections can be automatically applied to real-world software solutions by extending the software attestation model.

This thesis will report the author's contribution to the existing state of the art in the field of Software Attestation according to two primary levels.

High-level contribution. The dissertation will set the requirements for a valid attestation architecture by unifying the concepts found in literature and the hurdles encountered in practice. A generalisation of software attestation methods will define this technique as a robust abstract procedure. That is, the author will demonstrate that software attestation is independent of the nature of the protected software assets, from the kind of integrity evidence involved in verification and from the implementation details of the deployed procedure. Hence, the author will propose a new perspective by presenting the transversal features of the general software attestation model. These features apply to any actual instantiation of the abstract model and ensure that the attestation procedures are robust and rigorous. Moreover, such a general approach will result in a flexible architecture that can be adapted to the needs and extended to best suit any case of application.

Low-level contribution. The abstract model defines the architecture, protocols, and high-level procedures for software attestation, hence, it lets the actual methodology free to work at a lower level. The contribution in this way will demonstrate that actual instantiations of the abstract model differ by the detection strategy. That is, different instances will differ by (1) the nature of the assets for which integrity has to be guaranteed and (2) the kind of evidence used to prove assets' integrity. Hence, two main classes of software assets will be considered for protection. Structural software feature will be the foundation for static software attestation and behavioural aspects of the software will enable dynamic software attestation. These two instances of the abstract model will be presented by this work to report the author's contribution in (1) validating the applicability of the reference architecture, (2) assess the detection ability of the actual instances and (3) underline limitations and security issues of the two most representative implementations of the technique (i.e. static and dynamic software attestation).

Moreover, the author will demonstrate that software attestation is a valid tamper detection method, but it cannot provide comprehensive protection due to limitations and intrinsic weaknesses. However, the discussion will further validate

the abstract architecture. Indeed, limitation and weaknesses come from the actual software attestation instantiation and do not come from its abstract model. The capability of the presented model to be extended and combined with other protections will demonstrate that the design effort has been spent in the right way. Hence, the discussion will present Reactive Attestation: a novel technique that demonstrates how the tamper detection ability of software attestation can be coupled with the reaction capabilities of other techniques. The dissertation will exploit this techniques combination to demonstrate that software attestation represents a building block for sophisticated protection techniques. Not eventually, it is the exact purpose of ASPIRE[2], an EC research project in which the work presented in this thesis has been developed in its early stages.

Dynamic software attestation is the object of another significant contribution reported in this document. It will discuss a specific technique, i.e. Invariants Monitoring, as a possible way of attesting software behaviour and execution integrity. Many limitations and critical issues will be identified. Many attempts to overcome the discovered limitations will be presented and discussed, and some of them will be found to be very hard to be fixed even with significant effort. Finally and unfortunately, the contribution in this direction will present negative results: the technique revealed to be not secure and weak, thus not applicable for software protection purposes. Hence, this work should warn and discourage those who may consider deploying Invariants Monitoring to protect their software.

Bibliographic foundation

The preliminary parts of the architecture and the static software attestation methodology are unpublished. However, many parts of this thesis are the subject of published works. Hence, several concepts presented in the remainder of the dissertation are deliberately inspired by the following published articles:

- the early studies about Invariants Monitoring have been presented in "Remotely Assessing Integrity of Software Applications by Monitoring Invariants: Present Limitations and Future Directions", presented at CRiSIS 2017: Risks and Security of Internet and Systems pp [52];
- a deeper discussion about Invariants Monitoring have been reported in "On the Impossibility of Effectively using Likely-Invariants for Software Attestation Purposes", published in the Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications [53];

[2]https://aspire-fp7.eu

- the work about Reactive Attestation has been detailed in "Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks", presented at the SPRO '16 Proceedings of the 2016 ACM Workshop on Software PROtection [54];

Thesis organisation

The remainder of this document will report the author's work as follows:

Chapter 1 opens up the dissertation by extensively analysing the literature and explaining the background in which software attestation lays its foundations.

Chapter 2 introduces the abstract software attestation model that acts as a reference architecture for technique instantiations.

Chapter 3 shows how the general model can be characterised to monitor static and dynamic properties of software. It presents the requirements, the design and the security analysis about the specific attestation systems and their protection abilities. Moreover, the same chapter discusses the automatic applicability of the presented software attestation techniques.

Chapter 4 demonstrates that software attestation can be exploited as support for protection methods that need tamper detection.

Chapter 5 draws the conclusions about the dissertation work and proposes possible future directions for software attestation.

Acknowledgements

First of all, I would like to thank prof. Antonio Lioy for allowing me to work in the Computer and Network Security Group and working with such great people in all these years.

As said, part of the work reported in this thesis has been carried out along with the ASPIRE project. For this reason, I acknowledge the effort spent by other people in collaborating to achieve such excellent results (dedicated thanks will appear inside the thesis composition).

Furthermore, I have to thank Cataldo (Aldo) Basile for having been the best guide I could ever have wished, and for his steady support throughout the whole doctorate duration. Finally, I want to mention Daniele (Canavese) and Leonardo (Regano) for all the work shared and the good time spent together.

To all of them,
I raise the glass

Chapter 1

Background

This chapter outlines the state of the art in the field of software protection based on Remote Attestation.

Remote Attestation is a mechanism by which a software system proves its integrity by providing a set of evidence. The final decision about the integrity of the protected system is taken by a remote component that resides on a remote and trusted host.

Remote Attestation is then defined as a tamper detection technique; it is based on a trusted server that evaluates the integrity of a remote system by checking the validity of provided evidence. Coker *et al.* formally defined the principles and the general architecture for Remote Attestation regardless of the underlying technological implementations [15]. Figure 1.1 reports the general Remote Attestation architecture. According to work mentioned above, the main components of an

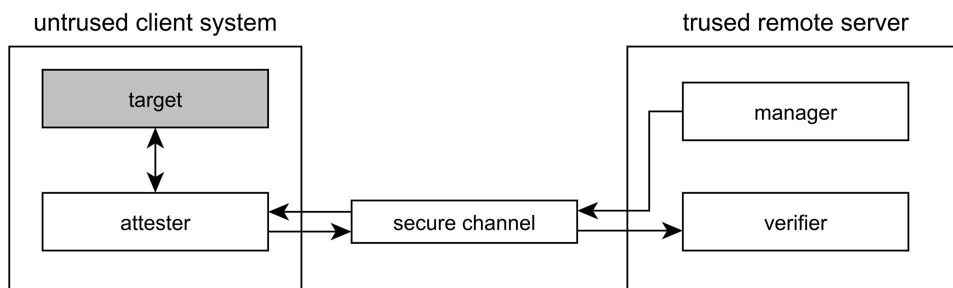

Figure 1.1: General Remote Attestation architecture.

attestation system are:

- **Target**, it is the final system that has to be protected, it is the set of software components whose integrity is monitored by the attestation mechanism;

- **Untrusted environment**, it is the environment where the target is deployed; in the MATE context, the end user is entirely in control of the execution environment and has full privilege on it; thus it cannot be assumed to be trusted;
- **Attester**, this component is in charge of extracting evidence to prove the integrity of the monitored system;
- **Manager**, it triggers the Attester to start an attestation procedure, thus to produce integrity evidence, from the trusted server;
- **Verifier**, this component receives the evidence collected by the Attester and evaluates them in order to decree if the target system is still in a valid state or not;
- **Secure channel**, all the communications among the components pass through public networks then, a protected channel is needed to avoid information leakage or Men-in-the-Middle attacks.

Given the general scheme of Remote Attestation, the first distinction that emerges depends on the practical implementation of the trust base and secondly on the nature of the collected integrity evidence.

1.1 Hardware

The early stage works about remote Attestation exploit hardware components as a trust base to build a robust root of trust. A critical event that enabled the diffusion of hardware-based Remote Attestation has been the specification of the *Trusted Platform Module (TPM)* by the *Trusted Computing Group* [25]. The TPM is a hardware component, a chip, and provides a set of security operations performed in a trusted way, such as cryptography primitives, keys generation and management, pseudo-random number generation and small storage space. Hence, the broad set of promising features and the relative expectations led the TPM to be the most exploited element as a trust base for Remote Attestation.

Several works have been proposed to attest the integrity of a running system in different ways but always exploiting the TPM. One of the first and one of the most influencing work has been presented by Sailer *et al.* [45]. In this work, the authors defined the *Integrity Measurement Architecture (IMA)* that aims at measuring the whole environment of a running machine, from the BIOS up to the operating system and the software it loads. Every software binary and configuration file whose integrity needs to be attested is measured; the obtained measure is signed using TPM facilities and stored for future requests from a remote appraiser. The integrity

measure is a hash of the attested element. In this way, every component of the target environment can be statically attested, and the whole system configuration can be verified.

This approach is intrinsically limited, statically measuring a component does not ensure that the obtained measure is valid when the component is used. Practically, there is a discrepancy between the *time-of-use* and *time-of-attestation* of any element, as stated by Shi *et al.* [48]. In order to mitigate this issue, the same authors proposed BIND. They tried to narrow the gap between time-of-attestation and time-of-use by moving the Attestation closer to the execution time. As soon as a process is launched, the attestation procedure starts as well and collects integrity measures from the input/output data and the loaded binary image.

Another issue that emerged from IMA, therefore from static files attestation, is that the bare measuring of files' integrity does not protect from non-static attacks. That is, attacks that do not alter the structure of the target's components are not detected. Jaeger *et al.* proposed PRIMA to enable measurement of information flow integrity [29]. The proposed solution implements the IMA architecture but uses different integrity evidence, i.e. the information flow. This solution is still attesting program structure, but from a behavioural point of view, i.e. information flow can be altered also without modifying binary or any other application related file.

A paradigm shift in the collected evidence represents a step further in improving the attestation's effectiveness. Chen *et al.* proposed to verify if the target system is respecting three main security properties classes [13]. *Code Control*, to enforce the machine to behave as expected; *Code Analysis* to enforce code and machine properties; *Delegation* to enable a trusted third party to certify the desired properties. Hence, they rely on a *a priori* analysis of the target to deduce the security properties that the protected system has to ensure. Once these properties are set, the Attestation acts as a challenge-response mechanism to attest execution against the given input. In practice, the provided input is used to perform computations on the attested system to verify that the set of previously deduced requirements (properties) are satisfied. The scheme uses a hybrid approach: a property attester, which runs as security kernel service, calls on a binary attester that exploits TPM facilities.

Following the way of changing that moved attestation evidence toward something different from structural properties, Kil *et al.* proposed ReDAS [32]. This attestation mechanism aims at monitoring the integrity of software applications through dynamic system properties inferred from global variables values collected at system call invocations. Then, they claim the system can offer protection from tampering. It represents a significant improvement for Remote Attestation as it gives importance to dynamic features as descriptive for system behaviour and execution.

So far, the dissertation presented the evolution in the literature about Remote

Attestation mechanisms in a classical scenario. Targets to monitor are (or run on) physical machines where a standard commodity operating system with the full software stack and configuration files gives the environment. As the technology advanced, new distributed architectures appeared, such as cloud environments and *Internet of Things*. In these environments, it is not possible to assume to have specific hardware devices like TPM to rely on or, at least, it cannot be assumed to reach it. For instance, in virtualised environments, it cannot be assumed that the TPM feature can be reached from inside the virtual guest environments.

According to evolution, the trusted computing has evolved as well and, consequently, the literature proposed solutions for virtualised environments. Garfinkel *et al.* proposed Terra, a virtualisation solution that exposes TPM features to every virtual machine independently through a virtual monitor that is integrated into the hypervisor [22]. This solution relies on the TPM that is supposed to be present on the physical machine that hosts the virtual ones. In the same way of Terra, Santos *et al.* proposed a Trusted Cloud Computing Platform that allows cloud environments to rely on trusted computing features and to be inaccessible from the underlying host system. Perez *et al.* proposed a software virtualised TPM in order to provide the same interface to virtual guest systems [40]. These works did not bring any new attestation mechanism; they instead demonstrated that Remote Attestation is a valid protection technique, even in today's' scenarios, and it just needs technological adjustments.

On the other hand, a set of works proposed alternatives to the TPM by exploiting different hardware solutions. Basile *et al.* proposed the use of reconfigurable devices (FPGA) as a hardware self-protecting trust base [8]. In their approach, Attesters are implemented by the FPGA cores. The cores can attest the executed binaries in memory by directly accessing them, thus bypassing the operating system. In the same way, Noorman *et al.* proposed Sancus, an architecture for resource-limited devices [38]. The authors rely on (minimal) hardware extensions to provide Remote Attestation and integrity guarantees. The hardware extension provides cryptographic facilities for keys derivation and digest computations, which enable Attestation of loaded executable code. The Sancus attestation scheme can verify the structural integrity of a software module and that it is loaded at the expected address in memory. The system performs verification by comparing the resulting digest with the expected, genuine one. In Copilot, Petroni *et al.* use an add-in card connected to the PCI bus to perform periodic integrity measurements of the in-memory Linux kernel image [42]. Then the trust base is implemented by an external device connected to the monitored system through the PCI bus. Both the just mentioned solutions still exploit static integrity measurements.

On the commercial side, Intel proposed the Trusted eXecution (TXT) Technology[1]. TXT is a hardware technology that provides Attestation of the authenticity of system software. Intel TXT exploits TPM and cryptographic techniques to measure software and platform components; it can be used for attestation purposes. Moreover, Intel proposed a complete Remote Attestation mechanism based on Software Guards eXtensions (SGX) [16]. SGX provides a set of extensions to the Intel x86-64 architecture that aims at providing security-sensitive computations in potentially malicious environments. Critical software pieces can be executed in isolated hardware containers, i.e. *enclaves*, which can contain sensitive data and code to manage it. Enclaves enable secure remote computation, which represents a fundamental building block to run trusted code that attests the target system. Hence, the SGX attestation system exploits enclave and pre-installed certificates to compute and sign attestation evidence on the target system securely. For the mobile world, ARM has developed the TrustZone[2] technology that partitions the processor into two independent cores. One of them is for normal operations, while the other one is for security-critical tasks. The secure processor is fully secured in hardware and completely inaccessible from the other one. The ARM TrustZone is the starting point of some hardware-based Remote Attestation solutions implemented by industry, e.g. Samsung KNOX technology[3].

In conclusion, hardware-based solutions brought Remote Attestation to be practically exploitable, and then they made this technique to be of interest for further investigations. Early works defined the general architecture and proposed basic methods to attest target structural integrity; after that, researchers have improved the original model to achieve better descriptive methods to model the target's integrity. Finally, industrials brought Remote Attestation in real-world applications, thus confirming the validity of the technique.

1.2 Software

In modern days, participants in distributed systems are no longer uniform, and the nature of the nodes of a network can vary very much. On the server side, there can be traditional monolithic machines running single operating systems and a set of services; physical hosts running multiple virtual machines that emulate traditional ones; multiple physical hosts that run several virtual machines, which provides single or multiple services, i.e. cloud environment.

On the client side, the scenario is even more diverse; the variety of devices

[1]https://www.intel.com

[2]https://www.arm.com

[3]https://www.samsungknox.com

that can be connected to the network is extensive. There can be regular personal computers running traditional OSes, personal mobile devices running different OSes (in different versions), modern smart devices with custom hardware and software such as domestic appliances, office devices, sensors. Therefore, modern networks are made of very different devices that may not be driven by a user, i.e. *Internet-of-Things*.

The evolution in distributed networks led to new attacks model as well. Due to a large number of connected devices and to the increment of their computational power, contemporary services follow the client-server paradigms that tend to delegate executions to client devices as much as possible in order to reduce the server load. Hence, client devices are often asked to execute code and to manage data that can be sensitive or of interest for malicious users. MATE attacks threat several assets. For example, intellectual property can be stolen to take advantage from others' procedures, software integrity and execution correctness can be violated to force arbitrary behaviours on applications, copyrighted contents can be abused and disclosed.

MATE attacks motivate the need for a remote integrity checking mechanism. Remote Attestation has been considered as a valid protection technique to manage the integrity checking needs. Due to the large variety of hardware involved, it is not possible to assume that a particular chip is available as done with TPM. Indeed, devices running the same program can have different underlying hardware (and even software) solutions. Consequently, it was necessary to find hardware-independent solutions on which a root of trust could be built. The aim is to achieve *Dynamic Root of Trust*, that is to trust the Attester, hence the target, on dynamic software properties. Consequently, several mechanisms have been presented in the literature to overcome the hardware limitations for Remote Attestation.

Sadeghi *et al.* gave a general definition: *Software Attestation* is a protection mechanism that aims at monitoring the integrity of a remote target without any hardware component to rely on to build root of trust [43].

Spinellis *et al.* proposed the earliest work on static software attestation; they used the hash of random parts of the memory as integrity evidence and a challenge-response protocol to build root of trust [49].

Seshadri *et al.* proposed Pioneer: one of the first attempts that do not rely on HW specific components for Remote Attestation purposes [46]. Indeed, they compute evidence using a checksum function and hashes of the result to send back for verification. To trust the client, they exploit the time spent to compute the integrity evidence, and if it exceeds a predefined delta, the evidence cannot be trusted. This early solution is not entirely hardware independent: the execution time of a procedure on the client is assumed to be related to the underlying hardware. Then, it is another way to attest hardware integrity, but it implies that the platform is known. Seshadri *et al.* previously proposed a similar solution, namely SWATT, in

which they exploited the same methods of Pioneer in embedded devices [47]. Another work that adopts a time-based solution to trust the client was published by Armknecht *et al.* [5]. Their method employs a challenge-response protocol; the random challenge sent to the Attester determines which memory locations have to be extracted for integrity evidence computations. The collected evidence is accepted by Verifier depending on the execution time, like the previously cited works.

A proposal that slightly shifted the paradigm of Attestation is MobileGuards by Grimen *et al.*, short-lived Attesters are downloaded from a trusted server [24]. The downloaded code is executed to decree the integrity of the target.

As it happened for hardware-based, in software attestation the need to attest dynamic properties has emerged. Static Attestation monitors static properties, which are properties about the structure of the target, e.g. binaries, executable memory portions, configurations, images and files in general. The main issue that arises from this approach is related to attacks that do not alter the structure of the target but tamper with it at runtime by altering its functionalities, e.g. system calls, functions redirections, return-oriented programming and cloning. Then, several works have proposed to monitor properties that are more semantically descriptive for the execution integrity. Practically, literature moved evidence collected for Attestation from something that describes the structure to something that describes the behaviour of the target. Therefore, software attestation started to monitor *execution correctness* instead of pure structural integrity. To pursue this goal, different kinds of software properties have been proposed by published works.

Sadeghi *et al.* proposed to focus on software properties instead of specific software or hardware component [43, 13]. *Property-based* Attestation aims at verifying whether target properties are sufficient to fulfil predetermined security requirements. This approach opened up the way towards dynamic Attestation and has been exploited to realise a virtual TPM and a property-based bootstrap architecture [44, 34]. The proposed solution allows classical Remote Attestation mechanisms to be applied even without any underlying physical device, but it does not introduce any novelty in terms of methodologies.

Property-based Attestation has some limitations. Indeed, it is hard to define properties that are related to trustworthiness. As an alternative to static and property-based Attestation, Li *et al.* tried to overcome this limitation by proposing *model-based Attestation* [36]. This work introduces a behaviour-based attestation model that tries to determine the trustworthiness of the attested target from its system related behaviours. The authors presented a model to describe system behaviour through a set of execution-involved parameters, which are the set of all the subjects, the set of executable programs, the set of inputs and the set of outputs. Alam *et al.* generalised model-based attestation in Model-Based Behavioural Attestation (MBA) [4]. They based the Attestation on evidence related to the behaviour of a policy model. Thus, the attestation feature is generalised and not specific. Moreover, the generalisation made the model suitable for any Remote Attestation

architecture. The goal of MBA is to provide a method to attest the usage model while accessing specific objects on the target.

Abadi *et al.* has proposed an alternative approach; they proposed to attest Control Flow Integrity [1]. The idea is to verify that transitions between software portions are followed with precise order and without sudden jumps. Hence, execution correctness and dynamic integrity are attested by verifying that the control flow of target respects the deployed one. The same concept has been recently re-proposed for low-end devices by Abera *et al.* [2].

An essential set of works for the remainder of this thesis exploit *likely-invariants* to model software structure and behaviour. Invariants are logical assertions that are always true for a portion or the whole application execution. These assertions are specified by the developer, during the preliminary design phases, starting from specification and requirements. Unfortunately, it has been demonstrated that manually defined invariants are likely to be invalid valid or absent in the final software application [20]. Consequently, researchers have developed tools to automatically infer invariants from applications, the most exploited in the academic literature is Daikon[4] [21]. The inference is mainly based on a statistical analysis of the application execution traces, and then, the deduced assertions may not always be valid. Indeed, given the experimental procedure that extracts likely-invariants, they can be considered valid only with a certain probability. On the other hand, the empirical nature of the inferred assertions itself made likely-invariants to be an auspicious feature to describe software patterns and behaviour.

Even if it is hardware-based, ReDAS by Kil *et al.* defined *Dynamic Attestation* (previously mentioned). It is the process of assessing the integrity of a target by checking its dynamic properties at runtime. Furthermore, they propose a dynamic attestation mechanism that exploits likely-invariants to check data and structural integrity at system calls invocation time [32]. They evaluate likely-invariants with values collected at runtime, thus decreeing the target correctness.

Baliga *et al.* proposed Gibraltar, a Remote Attestation system to detect kernel level rootkits in operating systems. They check the validity of the target data structures by evaluating likely-invariants against runtime values of crucial data structure [6].

In the end, software-based Remote Attestation has been introduced as a necessity to overcome technology limitations to hardware-based set by evolution. Despite the hardware-based one, it has been investigated more from the semantic point of view, than architectural and method one. As a result, it is possible to observe different ways to model the target's execution and to attest target properties that are more precise than the mere structural ones.

[4] https://plse.cs.washington.edu/daikon

1.3 Reaction

Remote Attestation itself acts as a tamper detection mechanism; then, it has to be supported by a reaction method. It means that, after an application has been decreed as no more sound by Remote Attestation, another protection mechanism must enforce countermeasures on the target to avoid an attacker to take advantage of the tampering.

To realise effective reactions, both locally and remotely techniques have been proposed in the literature. Even if reaction may seem the natural consequence of detection, tamper reaction has not been consistently explored. Indeed, the number of works about reaction is much smaller than the one about detection. Then, for what concerns the academic world, tamper reaction solutions have not reached a remarkable level of maturity or, at least, of diffusion. The industry, on the other hand, tamper reaction seems to be more advanced, but it is difficult to be demonstrated because of very few published works. It is common for companies to maintain their advances undisclosed in order to exploit as much as possible the so-called "security through obscurity principle".

The most trivial way of reacting could be to stop the target execution as soon as tampering is detected. This solution comes with a severe drawback, detection and reaction are performed at the same time. Consequently, it gives a hint to an attacker on where and when (in the code) the reaction takes place. Hence, this is not very effective because the effect can be directly related to the cause, i.e. the target crash is a direct consequence of the tamper. In practice, it is useful in stopping tampered targets, but it gives hints on how to disable the reaction components in the target.

To avoid trivial solutions that may threat the deployed protection technique itself instead of securing the target, Tan *et al.* defined basic principles for tamper reaction [50]. The most important principle states that tamper detection and tamper reaction must be separated in space and time. Components for detection and reaction must be deployed in a different location in the target and must be hard to tell from the rest of the application code. Moreover, the two elements must be executed in different time moments in order to decouple the cause from the effect, thus breaking down the detection-reaction relationship. In addition, the authors proposed to postpone the injection of software failures to slowly worsen the target functioning, i.e. to achieve *graceful degradation*.

Oishi *et al.* also proposed a graceful degradation reaction mechanism [39]. The original target binary is modified by replacing a set of instructions, namely the camouflage phase. The instruction is restored only at runtime only if the target is valid, that is the de-camouflage phase. If any suspect of tampering is detected, the reaction mechanism takes place and prevents the de-camouflage phase, thus hurdling the normal target execution.

Jakubowski *et al.* proposed an alternative to degradation and interruption: a

self-correcting system [30]. Distinct components are identified and made redundant; then a voting scheme is applied to results from the components copies. Moreover, diversification is used to make the code different even if the implemented functionalities are the same. Finally, a periodic verification is applied; a target checksum is computed at precise points in the execution flow. These three mechanisms allow the target to become tamper-resistant. If an integrity violation is detected, the protection applies delayed countermeasures, which are corrections of the results using redundant outcomes and encryption of working data.

All the proposed reaction mechanisms locally apply changes to the target in order to worsen or correct its behaviour. The changes made by the reaction inevitably reside on the attacker's (untrusted) environment, thus they could be eventually spotted. In conclusion, literature has not yet presented a valid reaction mechanism that is out of the end-user control, and that cannot be bypassed even if detected.

1.4 Open issues

Literature proposed several works such that Remote Attestation seems to have reached the right level of maturity, given its consistent level of diffusion. At the same time, many issues remain open and have to be discussed in order to improve the technique's robustness and protection efficacy.

The general architecture has intrinsic limitations that weaken the protection and have to be addressed. The main architectural limitation is bound to the Attester: this component has to run together with the delivered target. This limitation comes with all the different kinds of Remote Attestation, even with techniques that limit the lasting time of the Attester in the target system. In fact, for classical hardware-based Attestation, the Attester may be a process or a piece of code that is run when evidence has to be collected (e.g. at system bootstrap while BIOS or during the kernel loading phase). On the other hand, for software attestation, the Attester is a piece of code that is fully integrated with the delivered protected application; thus it is entirely under the user control that can inspect and manipulate it. Short-lived Attesters that are downloaded at runtime just limit the window of exposure but do not prevent attacks to the Attester itself. Indeed, the Attester is available and valid for a small amount of time, but its structure and behaviour can still be inspected.

Attacks that aim at removing or disabling the protection can be mounted against the different Remote Attestation techniques. Static techniques are vulnerable to memory copy attacks. An attacker can force the attestation system to check for integrity against an untampered version of the software that is actually run. In practice, the attacker tampers with the program and keeps a valid version of it. The correct version is used to perform Attestation on. Whenever an attestation request arrives, it is redirected toward the correct version instead of the tampered

one, which is still run apart.

A subtler version of this attack is performed by redirecting the attestation evidence extraction to a valid portion of the memory, as presented by Van Oorschot *et al.* [51]. Two copies of the target application are run at the same time, and all the Attester activities of the tampered application are redirected, employing memory pointer redirection, to the correct version. In addition, static attestation techniques cannot detect attacks that do not alter the binary structure of the target application, e.g. attaching a debugger to drive the application behaviour.

Time-based Attestation is vulnerable to proxy attacks. This attestation technique relies on the execution time for a precise task that directly depends on the hardware configuration. If an attacker runs the application on a system that is faster than the foreseen one, he can gain an advantage on the attestation procedure, i.e. continuous technological evolution of hardware strengthen this threat daily. In fact, in this way, the attacker can exploit the time gained to redirect the attestation procedure, to fake them or to retrieve attestation evidence from different places than the monitored target application.

Dynamic software attestation techniques are subject to false positives and false negatives. Dynamic procedures are somehow trained before the target application is run in real-world scenarios. Then, the trained procedures may fail if the target application reaches a state that has not been observed during the training phase, thus leading to a false positive. It means that the application is labelled as tampered even if no attacks are taking place. On the other hand, a set of attacks can pass unnoticed, i.e. false negatives. Dynamic attestation techniques exploit software artefacts to model the target application behaviour that may not suffice to describe software in all its states.

For what concerns software attestation, it is noticeable that the protection is particular and strictly related to the final platform on which the protected application will run. Indeed, the Attester has to: know how to extract evidence and to be fully integrated into the target. It makes the Attester the most critical element in the attestation architecture. It may be readily detectable due to particular low-level tasks performed to retrieve information and, in particular cases, due to is functioning itself.

In the next sections, a software attestation model is investigated from an abstract perspective to reason about the structural issues of the method. Consequently, research made spent effort in studying the ability of the abstract model in detecting the integrity violations on two main software properties, to the purpose of overcoming the limitations presented in this section and assessing the effectiveness of the protection technique. *Static software attestation* is investigated to monitor structural software features. *Dynamic software attestation* is studied as representative for all the possible attestation specialisations aiming at monitoring behavioural aspects of the software. For both the approaches, the reported study defines the

requirements and the security properties that drive the attestation model. The discussion finally presents the weaknesses and the points of strength of each attestation instantiation.

Chapter 2

Architecture

This chapter describes the design of a general architecture for software attestation systems. The work made in this direction firstly discussed the architecture from a theoretical point of view; hence, it defined a set of mandatory features for Software Attestation systems. Then, the dissertation presents practical directives to implements such theoretical requisites in practical procedures.

Here it is described the research effort spent in defining prerequisites and features that any software attestation mechanism should take into account to be practically usable. This work aimed at unifying the concepts from the literature; thus providing a general reference model for software attestation implementations. The author's contribution consists of the refinement of the existing architectural concepts as well as the collection of practical constraints and limitations. Moreover, the author research committed to formulating an abstract architecture able to demonstrate the applicability of the general software attestation concepts to any system, regardless of which software features monitored and protected.

2.1 Prerequisites

An abstract architecture definition is needed to set the requirements of a robust protection protocol for software attestation mechanisms.

Abstraction. The architecture definition has to describe the components, their interactions and operations with a high level of details such that it would be able to work regardless of the detection strategy that is chosen to implement, e.g. static or dynamic. It allows the software attestation architecture to be employed for any implementation that one may want to deploy.

Protocol. All the interactions that occur among components and the tasks that they perform have to be extensively defined and listed. Hence, the foreseen operations and their sequence must be unequivocally unique such that the software attestation system could not work in any other way than the described one.

Robustness. The design of the general architecture must not introduce vulnerability to any known attack. Hence, the abstract architecture must be well designed to prevent attacks that are not related to the deployed detection strategy, and that depends on the infrastructure. For instance, a software attestation architecture must be aware of replay, Man-in-the-Middle and sniffing attacks, which could be prevented by forcing challenge-based mechanisms (e.g. using nonces), client-server authentication and channel protections.

Attestation principles. The definition of a Software Attestation architecture has to be in line with principles defined by Coker *et al.* [15], thus avoiding inconsistencies that have been already studied and overcome.

Unpredictability. The general architecture must not involve repetitive mechanisms. Whether it is not possible to avoid repetitive mechanisms, they must be designed to prevent easily recognisable behaviour patterns. In other words, it is required that the attestation system cannot be easily understood, reproduced or faked by merely observing it.

Non-synchronicity. Attestation transactions must be initiated only by server-side components. Indeed, client-side triggered attestations would need to inject additional sensitive code and data in the target, thus enlarging the attack surface of the technique. Moreover, client-initiated attestation procedures would contradict the previous requirement (Unpredictability) as they would introduce repetitive or identifiable patterns.

Scalability. The design of the general architecture must consider that a large number of protected targets may run at the same time. Hence, the architecture must work without any degradation in the execution of both client-side and server-side components. Indeed, real-world applications can be distributed in a high number of instances that must be consistently attested at the same time. Moreover, the design must let the server-side architecture to scale in a distributed manner so that the relative deploy can exploit modern paradigms that natively enables scalability (e.g. cloud-based deploy).

2.2 Overview

This section provides an overview of the components involved in the software attestation architecture and their roles, the workflow of the general scheme and the assumptions that lie behind the performed design.

Assumption. The target application is supposed to be fully connected and to depend on network-based services exposed by a remote server. Then, the target can be split into two parts, a client-side logic that interacts with a server-side logic to work correctly and to provide the full functionalities to the final user. Although this assumption is not mandatory for applicability, it makes the protection system much more effective. A non-connected target application allows an attacker to bypass the monitoring system by simply disconnecting it from the network while it leaves the target business functionalities intact.

The software attestation system works alongside the client-server architecture of the target application and is composed of three fundamental components: the *Manager*, the *Attester* and the *Verifier*. In addition, two other components are foreseen in the proposed architecture: the Reaction Manager and the Reaction Enforcer. In the general architecture, software attestation works as follows.

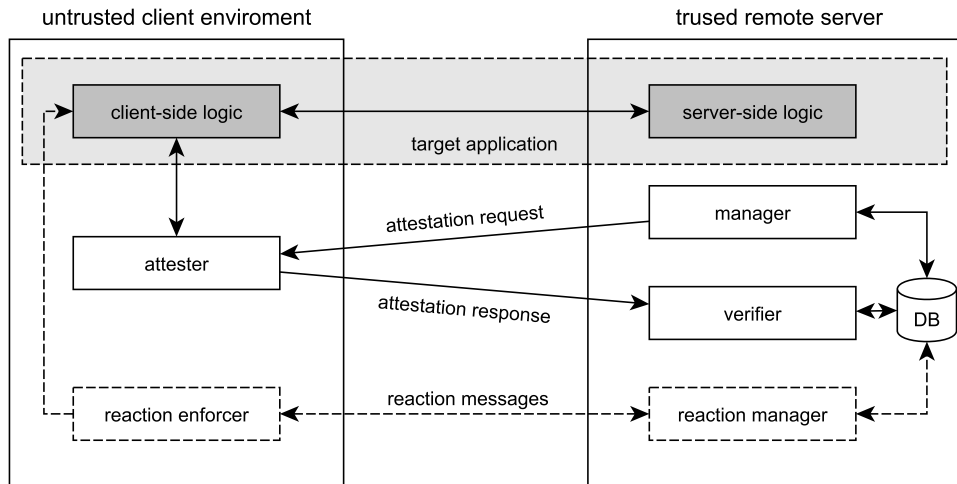

Figure 2.1: Software attestation architecture.

1. As soon as the target application starts, the Attester connects to the Manager and notifies the start-up event. Then, the Manager creates a session for the just connected instance and stores it into the database. The session is used to record information about attestation transactions and to schedule attestation requests. Afterwards, if the connected target needs to be attested at start-up, the Manager sends an attestation request and then schedules the next one.

2. During the target application execution, whenever it is necessary, the Manager prepares and sends an attestation request to the Attester. The most important element inside the attestation request message is a pseudo-random *nonce* that will be involved in the computation of the returning results. It helps to avoid replay attacks: an attacker that can intercept and store a valid attestation response could reuse it to reply to any future attestation request, thus presenting a tampered application as a good one.

3. The Attester performs the attestation routine to extract integrity evidence and sends them back to the Verifier. The attestation evidence is collected according to hard-coded routines and, eventually, driven by the nonce value. The nonce does not bring any information itself, but its value can be used as a seed for random procedures or to parametrise computations in order to make the extracted evidence unique for each attestation request.

4. The Verifier analyses the received data and, consequently, emits a verdict about the target application integrity. Then, the verdict is stored in the database, coupled to the target instance and the current session. Then, the database completely tracks all the attestation transactions.

5. The Reaction Manager asynchronously accesses the database to monitor the state of the target and its evolution. Whenever a violation in the target's integrity is detected, the Reaction Manager can decide whether or not to enforce countermeasures. Reactions may vary from very simple, drastically punishments for single violations, to very complex, rules sets punish the target according to sophisticated policies. Reactions may be enforced on both server-side and client-side.

6. The Reaction Enforcer waits for commands from the Reaction Manager to apply client-side reactions. The Reaction Enforcer practically applies decisions taken on the server-side by the Reaction Manager through predefined hard-coded procedures or by a dynamic interpretation of the received commands.

The following sections give a detailed description of all the features of the architecture's components and their workflows.

2.3 The Manager

The Manager is the server-side component that is in charge of starting each attestation transaction. Hence, whenever it deems, the Manager generates a pseudo-random nonce and sends it to the Attester. Although it is enough to trigger the

attestation procedure, it does not suffice to manage the target application life cycle. Hence, the Manager also takes care of recording all the generated attestation requests into the central database so that every attestation transaction is uniquely identified and stored for analysis purposes. Furthermore, the Manager manages the attestation requests scheduling. The central database records every deployed Attester together with the information about the relative required attestation period. The attestation period represents the average time between two subsequent attestations. Indeed, attestation requests must not be sent with a fixed schedule, as an attacker could predict the next request, restore a correct copy of the application, let it reply to the attestation request, and restart the tampered version of the application. Therefore, the Manager randomly selects the time of the next attestation procedure so that the average time between two attestations is kept constant with a tolerance interval. The desired average period value can be changed by any server-side component at any moment, thus allowing to alter the attestation frequency while the target is running, e.g. for reaction purposes.

To perform its activities and to manage different connected Attesters, the Manager is a two-tier component, composed of a single Master Manager and several Slave Managers. The Master Manager is in charge of performing preliminary operations for newly connected targets and of assigning each target to a Slave Manager. The Slave Manager performs the attestations.

2.3.1 The Master Manager

The Master Manager starts running together with the server-side environment; hence, it is always listening for new incoming connections from Attesters. When a client connects, the Master Manager reads from the database the information about the attestation frequency and creates a new session for the current target instance. Based on the read attestation frequency, the components estimates the effort required to attest the client and assigns the target application instance and its connection to the proper Slave Manager. Then, the Master Manager passes the connection with the client to the selected Slave Manager. After having passed the responsibility of the target application to one of the Slaves Managers, the Master Manager will ignore all other communications from the client.

The reference architecture can scale on a large number of server machines and for a large number of target applications as well. Indeed, while the Master Manager works a single server endpoint, the number of Slave Managers can be customised and adapted to the available resources. Moreover, this two-tier structure allows the Manager to be distributed on several different network hosts, that is, to scale up well and afford numerous targets. Several Slave Managers can run on different network nodes, and a single Master Manager can act as a dispatcher or different Master Managers can operate from different hosts and manage a dedicated Slave Managers set each.

In addition, it must be taken into account that every Slave Manager can manage more than one client, as the next section will explain.

Figure 2.4a reports the Master Manager workflow. As soon as the execution begins, the Master Manager initiates the Slave Manager tread pool. The number of the spawned Slave Managers is driven by the maximum number of threads that the processor of the hosting machine can offer, i.e. one Slave Manager for each thread. A higher number of spawned Slave Manager would result in an overload, thus preventing some of the Slaves to work correctly, that is to accomplish the requests schedule right on time. Then, it enters an infinite loop in which it waits for new client connections, fetches information about the just connected client from the database, selects the less loaded Slave thread in the thread pool and passes the connection to it.

Finally, the Master Manager acts as a dispatcher for the Slave Managers. It provides a single endpoint which clients address new connections to, which manages the connection setup and a set of Slave Managers that perform the actual attestation tasks.

2.3.2 The Slave Manager

This subcomponent is the one that performs the actual Manager's work for attestation purposes. It keeps track of the currently served clients, their schedule and the status of their connection. For this reason, the Slave Manager is more complicated than the Master.

The Master Manager spawns each Slave Manager as a single independent thread. During the normal working of the global Manager system, there are multiple Slave Managers running and each of them serves multiple target application clients. The choice to generate as many Slave Managers as the system cores has been made for scalability purposes. Multiple Slave threads permit to balance the effort needed for attestation over all the resources of the server-side hosting system. A single thread for Slave Manager would have meant to have a single core (or virtual core) of the hosting environment involved in serving all the clients, thus leading to an unbalanced situation: one core is overloaded and all the remaining ones are not working at all (for attestation purposes). It would represent an issue when the number of served targets increases. When the Manager system is not loaded, the clients are balanced over the Slave Managers threads that spend most of the time in sleeping. Hence, in this case, the operating system can schedule multiple Slave Managers threads on the same core. Indeed, the threads-to-core association is not forced in any way by the Master Manager. On the other hand, when the load increases, the number of served clients gets higher and the sleeping time of each Slave Manager decrease the operating system can distribute the threads all over the cores to best exploit machine resources. It allows the Manager system to scale up well, thus serving many clients as they were few.

As soon as the Slave thread starts, it initialises a set of data structures: the served client list and the scheduling list. The first one keeps track of all the currently connected clients that have to be server with attestation requests. For each client, this list stores the information to manage the session and to identify the client itself: client identifier, the specified average period between two subsequent attestation requests avg_c, the allowed tolerance for the attestation period var_c and the session identifier. All these data are used whenever a client needs to be attested to retrieve information from the database, create attestation requests and schedule the next attestation. The second list is used to schedule all the client for the next attestation. A target, i.e. its Attester(s), connects in an unpredictable moment and could have a different attestation average period, then the scheduling list sorts the connected clients depending on the time of the next attestation. The scheduling list implements a differential time scheduling: each element of the list stores the amount of time to wait from the moment its extraction from the list and the moment to send it the attestation request, namely *Time To Sleep (TTS)*. The absolute time to wait (TTS_{abs}) before an attestation request is sent to a client in the list is equal to the sum of the TTS of all the previous clients in the list plus its TTS. Hence, the first element in the list reports the next client to attest and the time to wait for doing it. A possible situation of the scheduling list is depicted in Figure 2.2 as an example over a timeline. To summarise, the scheduling list contains one element for each currently served client, every element in the list stores a differential time that represents the time to wait before sending an attestation request, starting from the extraction of the element itself.

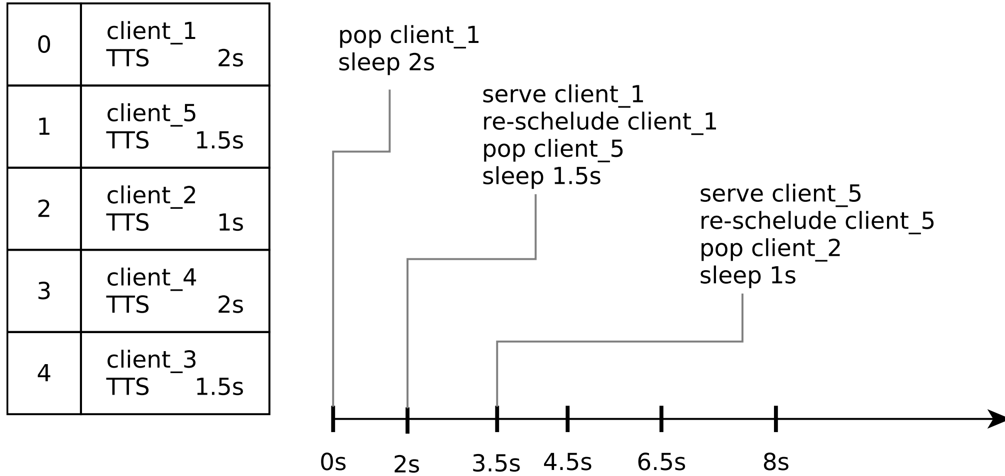

Figure 2.2: Scheduling list insertion.

At insertion time, a client comes with a TTS_{abs}, which is the absolute time to wait for the next attestation. Then, the list gets explored from the first to the last element, until the sum of TTS values of the already explored elements is less than

TTS_{abs} of the client to insert. In other words, the element is inserted in the last position that makes the sum of all the previous TTS less than TTS_{abs}. Once the position is identified, the actual TTS of the client to insert is computed. Then, the TTS of the subsequent element in the list is updated, and the element is inserted. Figure 2.3 reports an example of the insertion task for the scheduling list.

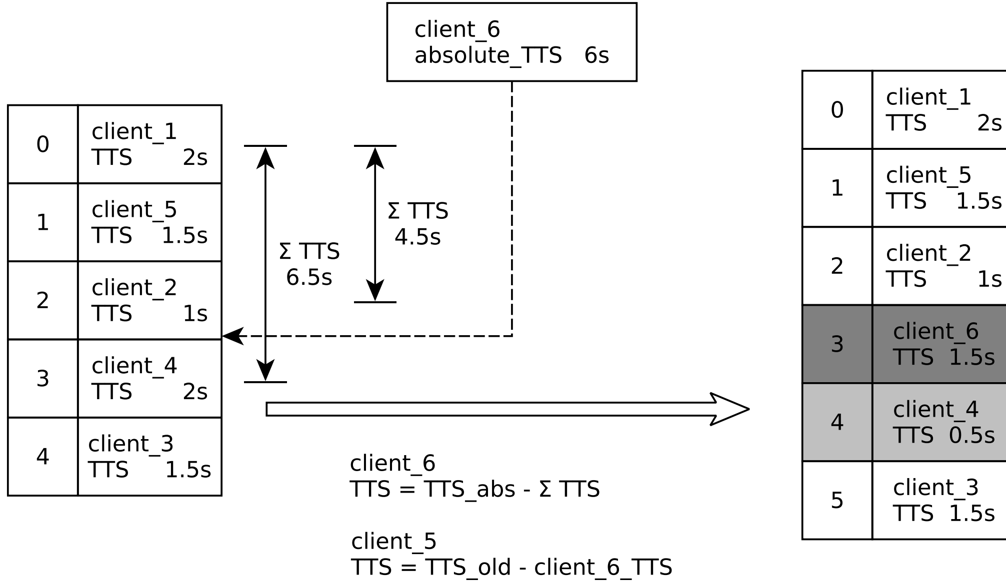

Figure 2.3: Scheduling insertion.

After the initial data setup, each Slave Manager enters an infinite loop as depicted in Figure 2.4d. This loop performs all the tasks needed to realise the actual Manager behaviour. First of all, it extracts the first element in the scheduling list and retrieves the TTS value. Then, it enters executes a sleep function: the Slave Manager enters an idle phase that can finish either because the sleep time is up or because a *new_client* event has been triggered. In practice, the sleep function is implemented using a conditional timeout wait for the *new_client* event thus enabling two different scenarios: the timeout elapses and the event does not take place, the *new_client* event happens and the timeout is not reached. The statement at line 7 in Figure 2.4d implements these two cases.

Then, if the timeout expires, it means that the extracted client has to be served. The Slave Manager generates a random *nonce*, envelopes it in an attestation request message, stores the request's information in the database and sends the request to the Attester. The sending of the attestation request allows the Slave Manager to detect whether the client is still connected or not. If yes, the client is inserted again into the scheduling list with a new TTS_{abs} that is calculated as:

$$\varepsilon = var_c - (rand() \mod (2var_c))$$

$$TTS_{abs} = avg_c + \varepsilon$$

On the other hand, if the client has disconnected, the session gets closed and persisted on the database. Consequently, the Slave Manager removes the client from the list of the served ones and starts the loop again.

If the sleep time is interrupted before it reaches the timeout and the *new_client* event is caught, the thread manages the new incoming client. The Slave Manager reinserts the client that was supposed to be served at thread wake up in the *scheduling_list*. The new TTS is equal to the value it had at extraction time, less the time slept (assuming the time to process the new client is negligible). Hence, the new client is inserted into *served_clients_list* (Master Manager already valorised all the data needed for attestation). After the initial managing of the incoming client, the thread checks if the client needs to be attested at start-up. In case of required start-up attestation, the Slave Manager prepares and immediately sends all the needed attestation requests to the client, otherwise, the client is inserted into the *scheduling_list*, and the loop continues with the next iteration.

2.4 The Attester

The Attester is the only client-side component in the attestation architecture deployed to the purpose of this document. This element is wholly integrated into the target binary and works beside the target business logic code; it starts executing before the target and runs independently from the target logic. This component aims at running in the background, waiting for attestation requests. Whenever a request is received, the Attester performs all the tasks that lead to extract integrity evidence to verify the integrity of the target. The kind of evidence extracted by the Attester depends on the deployed detection strategy.

The Attester can be included more than once in the target application, thus allowing to protect the application with different software attestation flavours or with multiple instances of the same software attestation version. Including multiple Attesters and multiple attestation techniques in a protected application allows to increase the level of security and to increase the attack time. Indeed, an attacker that wants to circumvent the software attestation features has to block or bypass all the included Attesters. Moreover, the Attester's instances inserted into the application can be configured to protect one another, thus making attacks much more difficult.

The Attester usually needs data to work correctly. The architecture is designed to include a precomputed binary object of data into the target such that the Attester can parse it and load the information needed. The structure of the included data strictly depends on the deployed detection strategy. Hence, all the operations needed to obtain the binary to inject into the target and to parse it when the Attester starts are delegated to the selected software attestation implementation.

To this purpose, additional tasks are performed during the target compilation: the data structures required by the Attester implementation are populated, encoded and embedded in the final binary so that they will be available at runtime. For instance, static attestation needs to know the binary memory areas that it has to monitor and their location at load time, invariants-based software attestation needs to know variables and their location in memory.

The workflow of the Attester lists a set of generic steps that do not depend on implementation choices. Then, Figure 2.4b depicts the overall behaviour of the Attester. As soon as the target starts, the Attester starts as well and performs necessary setup tasks. Data structures are parsed and loaded (if the detection technique requires it), the connection with the Manager is established, and all the connection parameters are stored in a session object for future communications. Whether any setup activity fails, the Attester itself and the target application are prevented from starting their routine.

If all the preliminary setup steps do not fail, the Attester begins an infinite loop that only terminates if the entire target application stops. The main loop waits for attestation requests coming from the Manager. The receiving of a request triggers the collection of integrity evidence. The Attester extracts from the running application all the data needed for attestation, and stores them in a buffer. The nature of the extracted data and the procedure to extract them are tightly-coupled with the kind of software attestation to realise. Then, the `extract_data` function (reported in Figure 2.4b at line 8) has to be implemented according to that choice and has to return a buffer that contains the data to send back to the Verifier. A keyed digest function receives the data buffer containing the attestation evidence and computes a signature on that data using the received nonce. It prevents replay attacks and strongly relates the collected data to the attestation transaction that is being performed. Then, the concatenation of attestation evidence and its digest provides the final data that will be sent to the Verifier. After that, the Attester connects to the Verifier and sends the obtained buffer. Finally, the loop starts again by waiting for another attestation request.

In practical terms of the delivered architecture implementation, the Attester is a software component that is compiled together with the target application code. It is a static library, and it is linked with the target application at linking time. The Attester static library has a single entry point, i.e. a function, which is invoked as soon as the target software starts. The operating system invokes the Attester entry function before the main function of the target or, in case of a shared library, before the library load. This behaviour is achieved through compiler's attributes assigned to the Attester entry point function, e.g. `constructor` for GCC. Then, two different scenarios could be possible: the target software is a standalone application or a dynamically linked library. In the first case, the Attester entry function is invoked by the operating system before the application's main function starts. In the second case, the Attester function is invoked at the library loading, before the requested

library function is executed.

2.5 The Verifier

The Verifier is the server-side component that checks the validity of the evidence provided by the Attester. Like in the case of the Manager, this component has to manage multiple connections from different clients (i.e. Attesters). In addition, the evidence data format and the implementation of the Verifier strongly depend on the deployed software attestation and on the chosen parameters Consequently, every Attester has to communicate with the proper Verifier. Then, a set of different Verifiers is available on server-side. For this reason, the Verifier is made up of two subcomponents as well: the Attestation Response Dispatcher and the Actual Verifier. Unlike the Manager, the two subcomponents are implemented as two separate processes. In this way, the Actual Verifiers can be separately compiled according to the chosen attestation version and parameters.

2.5.1 The Attestation Response Dispatcher

This subcomponent is the actual connection endpoint for all the Attesters. That is, every new connection is addressed to this component. This component is stateless, i.e. connections are independent one another as stateful information is kept on the database by the Manager.

Whenever a connection from an Attester comes, the Attestation Response Dispatcher elaborates the received attestation response to determine which target (then, which Attester) has connected. Then, the Attestation Response Dispatcher retrieves from the database the parameters of the request and the target. As described before, the Manager previously stored this information at request generation time. Then, the Attestation Response Dispatcher retrieves the nonce used to create the request associated with the received response. The nonce is used to verify the digest of the received data, thus to prove they are freshly generated. Whether the digest does not correspond to the expected one, the attestation transaction failure is recorded into the database, and the verification procedure stops.

Once the hash has been decreed to be genuine, the Attestation Response Dispatcher identifies which Actual Verifier binary is associated with the sender Attester. Then, Attestation Response Dispatcher launches the Actual Verifier and passes it the attestation data, and finally restarts the loop.

2.5.2 The Actual Verifier

This subcomponent verifies that the provided integrity evidence is valid and that the target application is still sound. The internal details of this element are related

to the actual software attestation implementation and its parameters. Then the Actual Verifier can be significantly different depending on the system conditions. Hence, this component's details will be discussed in the next sections for each different software attestation techniques that will be presented in the document. Anyhow, the general workflow of this component is quite trivial: it extracts each attestation evidence from the attestation data received in input and uses them for peculiar computation and comparisons as reported. This component takes the final decision about the single attestation transaction and stores it in the database according to a predefined transaction status (see Section 2.7).

2.6 The Reaction Manager and the Reaction Enforcer

The definition of software attestation does not include components for reaction decision and enforcement as the technique is only meant for tamper detection. However, the purpose of this work was to bring software attestation to be a comprehensive protection technique that can be worthy of being used alone. To this purpose, detection is not sufficient, and the software attestation architecture has to include reaction components.

Given that the detection is based on a client-server paradigm, the reaction must be designed according to this scheme. Hence, two components have been foreseen in the architecture: *Reaction Manager* and *Reaction Enforcer*.

The Reaction Manager is deployed on the server side. Given an application, the Reaction Manager is in charge of monitoring the attestations history for each session to identify anomalous behaviours. In practice, the Reaction Manager accesses history of stored attestation request and associated responses to decide whether to apply a reaction or not. The Reaction Manager is an abstract component that can be implemented in different ways in practice. For instance, reaction decision can be very straightforward as stopping a target as soon as tampering is detected or very complex like involving policies or decision strategies. Sophisticated decision strategies may involve the history of attestations within a session or, even more, across multiple session. Indeed, a Reaction Manager may implements policies to stop or degrade the execution of a target to punish it whenever attestations fail multiple consecutive times or simply to reduce the attestation period to monitor the target more strictly. Reactions to punish targets may be applied directly at server-side, e.g. attestation period reduction, or may need to be applied at client-side, e.g. execution stop or degradation.

Given that it is on the server side and it does not have direct access to the running target, the Reaction Manager is not able to apply reactions on the target, on the client-side. Hence, the Reaction Enforcer is in charge of apply, on the target, the decisions taken by the Reaction Manager. The Reaction Manager and the

enforcer communicate using the network and, preferably, using the same connection used by the other software attestation components. It would reduce the possibility that reaction messages could be told from the attestation ones. As for the reaction decision, the reactions that can be applied may vary. Indeed, execution stops can be caused by injecting runtime faults, deleting parts of the loaded binary code, reference errors and many other software failures. The same is valid for execution degradation. Then, the internal details of the Reaction Enforcer depend on the implementation choices taken while that reaction system is designed to be applied practically. Moreover, the implementation of the Reaction Enforcer is bound to the Reaction Manager implementation. The Reaction Enforcer has to know how to interpret the received commands and how to act accordingly.

Besides, these two components are both optional for the reference architecture. The Reaction Manager may be useless if the implemented reaction is triggered locally, without any remote decision. For instance, the Reaction Enforcer may apply reactions based on decisions taken directly on the target when failure attestations are detected. On the other hand, the Reaction Enforcer may not be included in practical implementation if reactions are ultimately decided and enforced on the server-side. It is the case in which, for instance, target logic depends on server-side computations that can be prevented by Reaction Manager (as for the solution presented in Section 4).

Finally, reaction mechanisms are foreseen by the design of a general reference architecture to enable software attestation to detect and react to tampering, but the actual details of the reaction components strictly depend on implementation choices.

2.7 The central database

The central database plays a crucial role in the functioning of the whole attestation mechanism, regardless of the actual technique that is implemented. As described in the previous sections, the database stores the data to manage all the processes related to the protection mechanism.

Applications. First of all, all the protected applications are listed in a table, namely *Application table*. Hence, every newly protected application is assigned a string that uniquely identifies it in the software attestation system, as depicted in Table 2.1.

Attesters. In addition to the applications list, the database lists also all the Attesters per single target application in the *Attesters table* (Table 2.2). Indeed, whenever a Attester is built to be inserted into a target application, it receives an incremental identifier that is unique for the protected application. Together with

this information, the Attester table stores the data to manage the scheduling of the client by the Manager, i.e. avg_c and var_c.

Attest-at-Startup assets. The assets monitored by each attester do not need to be listed in the database as the object of attestations is randomly selected by the request nonce. The only information about assets the database stores is related to the attest-at-start-up procedures. In the *Attest at Start-up table* (Table 2.3) every line report the Attester ID in the Attester table and a asset label. Then, each line of this table refers to an asset that requires an attestation request to be sent as soon as the target starts.

Sessions. To manage the application life cycles and attestation transactions, a set of tables have been introduced. The *Session table* (Table 2.4) records all the session for each Attester, then all the Attesters are treated independently, regardless of the other ones in the same target application. Along with which Attester the session belongs to, each record stores the begin time, the end time and the status of the session. The Manager creates a record in this table when an Attester contacts it and the session starts. The `start` field is valorised with the current timestamp and the `isActive` field is set to true. When the client Attester disconnects, the `finish` timestamp is recorded and the `isActive` field is set to false.

Requests. The *Request table* (Table 2.5) stores all the attestation transactions and relates them to the associated session. A record is inserted by the Manager every time a new attestation request is generated and sent to a Attester. After that, a record in this table can be modified only by the Verifier whenever an attestation response is received. Hence, every record in this table describes an attestation transaction and associate it to a session. Every record is created with the `sendTime` field valorised with the proper timestamp and with the `status` field that refers to the `PENDINIG` record in the Status table. In addition, every request is associated with a validity time, which is the maximum interval between the request sending and the response receiving for which the response can be considered still valid. If the response time exceeds the interval, the received attestation response is nevertheless processed, but the attestation transaction is labelled as expired. The `isStartUp` field tells whether the attestation request was sent at target application launch time or during its normal execution runtime.

Statuses. Every request has its life cycle, then, in order to describe it, a set of statuses have been designed and stored in an enumerative table, namely the *Status table* (Table 2.6). The design preferred to use this representation rather than a static enumeration of statuses to make the implementation flexible and extensible

in the future. Indeed, it is possible to introduce new statuses, in case it is needed, by simply adding a line in the Status table.

The foreseen statuses for an attestation transaction are the following ones:

- `PENDING`, this status is assigned to a request as soon as the Manager creates it. It happens when a request has been sent to the Attester, but no responses have been received yet. It is the only status the Manager sets, all the other statuses are assigned by the Verifier.
- `SUCCESS`, the Verifier associates a request record with this value whether an attestation response is received within the validity time, and the target application is evaluated to be genuine.
- `FAILED`, the Verifier associates a request record with this value whether an attestation response is received within the validity time but the integrity checks performed on the provided evidence gave a negative result.
- `EXPIRED_SUCCESS`, a request is referred to this value if the relative response produced a valid integrity check but out of the validity time.
- `EXPIRED_FAILED`, a request is referred to this value if the relative response failed the integrity check and it came out of the validity time.
- `EXPIRED_NONE`, this value is associated with a request for which a response has never arrived.

Column	Type	Description
id	int	Primary key for the table
application_id	char(32)	A 32 characters long string ID to identify the application
description	varchar(MAX)	An optional string to describe the application

Table 2.1: Application table description.

Column	Type	Description
id	int	Primary key for the table
application_id	int	Foreign key to the target application
attesterNo	int	Identifier of the Attester for the target application
description	varchar(MAX)	An optional string to describe the Attester
sleepAvg	int	Average attestation period
sleepVar	int	Attestation period tolerance

Table 2.2: Attesters table description.

Column	Type	Description
attester_id	int	Foreign key to the Attester (Attesters table)
asset_label	smallint	Label of the referred asset

Table 2.3: Attest at Startup table description.

Column	Type	Description
id	int	Primary key for the table
attesterID	int	Foreign key to the Attester (Attesters table)
start	timestamp	Timestamp relative to the moment when the attestator connects to the Manager
finish	timestamp	Timestamp relative to the moment when the attestator disconnects from the RA manager
isActive	bit	States wether the session is currently active or not

Table 2.4: Session table description.

Column	Type	Description
`id`	int	Primary key for the table
`sessionID`	int	Foreign key to the associated record in the Session table
`sendTime`	timestamp	Timestamp relative to the moment when the request has been sent
`responseTime`	int	Number of seconds between the request sent and response receival
`status`	int	Foreign key to the response in the Status table
`validityTime`	int	Time within which the response has to arrive to be considered valid
`nonce`	byte(32)	The random nonce included in the request
`isStartup`	bit	True if the request is relative to a startup attestation, false otherwise

Table 2.5: Request table description.

Column	Type	Description
`id`	int	DB id for RA request status
`name`	int	Enumerative name of the request status
`decription`	timestamp	Optional description of the status value

Table 2.6: Status table description.

```
    Data: scheduling_list, serving_thread_pool
 1  begin
 2      scheduling_list ⟵ ∅;
 3      for t ∈ serving_thread_pool do
 4          initialise t with servingRoutine;
 5      end
 6      while True do
 7          wait connection from Attester;
 8          read client parameters from DB;
 9          initialise session in DB;
10          select least loaded Slave thread;
11          notify new_client event to thread_pool;
12      end
13  end
```

(a) Master Manager algorithm.

```
    Data: a, d, app_ID, session, vars_list, n
 1  begin
 2      parse ADS from memory;
 3      connect to Manager;
 4      save connection parameters into session;
 5      while True do
 6          wait attestation request;
 7          nonce from request;
 8          data ⟵ extract_data(nonce);
 9          digest ⟵ digest(data, nonce);
10          a ⟵ data||digest
11          connect to Verifier;
12          send a to Verifier;
13      end
14  end
```

(b) Attester algorithm.

```
 1  begin
 2      while True do
 3          wait for a connection from Attester;
 4          receive attestation response r;
 5          identify the target from response;
 6          read session parameters from DB;
 7          verify attestation data hash;
 8          if hash not verified then
 9              record the event in the DB;
10          else
11              select the proper Actual Verifier;
12              launch the Actual Verifier with
                  attestation data;
13          end
14      end
15  end
```

(c) Attestation Response Dispatcher algorithm.

```
    Data: scheduling_list, served_clients_list,
          wake_reason ∈ {timeout, new_client}
 1  begin
 2      wake_reason ⟵ timeout;
 3      while True do
 4          client ⟵ remove first of
              scheduling_list;
 5          time_to_sleep ⟵ client's time to sleep;
 6          timeout-wait(time_to_sleep) for
              new_client event;
 7          if wake_reason = timeout then
 8              nonce ⟵ random bytes;
 9              store nonce for session in DB;
10              send att_request(nonce) to client;
11              if client disconnected then
12                  close session for client;
13                  remove client from
                      served_clients_list;
14              else
15                  insert client into
                      scheduling_list;
16              end
17          else   /* wake_reason = new_client */
18              insert client into scheduling_list;
19              insert new_client into
                  served_clients_list;
20              if attest_at_startup(client) then
21                  nonce ⟵ random bytes;
22                  store nonce for session in DB;
23                  send att_request(nonce) to
                      client;
24                  if client disconnected then
25                      close session for client;
26                      remove client from
                          served_clients_list;
27                  end
28              else
29                  insert new_client into
                      scheduling_list;
30              end
31          end
32      end
33  end
```

(d) Slave Manager algorithm.

Figure 2.4: Algorithms of software attestation components.

Chapter 3

Detection

This Chapter describes the research effort invested in designing a software-based system able to work on real-world applications according to the architecture previously defined (Chapter 2). The aim is to model a system able to detect violations of software integrity for both static and dynamic software features.

The general definition of the protection scheme permits to monitor the integrity of a large variety of software features. The software features to be used by the designed software attestation have been selected by considering two kinds of attacks that the technique has to be able to detect. The nature of attacks that can be ported to a piece of software mainly involves static or dynamic modifications.

In the first case, an attacker tries to alter the normal behaviour of the application by tampering with its structural properties, thus forcing the application to work differently by modifying the binary layout of the target. For instance, a static attack can be ported by removing a function call in the compiled application by merely replacing the relative `CALL` instruction with the `NOP` instruction. Then, it is possible to circumvent security checks, such as license verification, or to prevent any other functionality of the original application. The same is valid for data embedded in the software binary that can be altered to drive the software behaviour arbitrarily.

In the second case, the attacker tampers with features that are not related to the software structure, rather with features that depend on the program execution such as variable data in memory or execution flow. An excellent example of this kind of attacks is represented by debugging attacks, in which the attacker attaches a debugger to the running application and arbitrarily drives the program execution. Attackers may bypass function calls or alter and steal data from memory remaining unnoticed from a static point of view. All the possible modifications that can be ported on dynamic software features cannot be detected by merely monitoring the software binaries.

It is then clear that a software attestation system aiming at comprehensively detecting as much as possible integrity violations have to be able to monitor both static and dynamic features of a software. Consequently, two different software

attestation implementations have been pursued to obtain two software attestation models.

3.1 Preliminary requirements

Software attestation is meant to detect integrity violations on software binaries. Hence, it is mandatory to define a set of requisites to design an accurate detection mechanism for software attestation. The requirements hereafter discussed must be considered as an addition and, in some cases, as a specification of the ones presented for the overall architecture (Chapter 2). In other words, these requirements are reported in this chapter as their specification focuses more on procedures than on architectural schemes or structures.

Assets. The definition of the assets to protect has to be the first task in designing and implementing a protection technique. Indeed, it is necessary to know what the technique is going to protect to design the best protection strategy. For instance, for static software attestation, the assets would be tied to software binary features, thus to its executable code. Hence, any instance of static software attestation has to define: the target parts that are intended as assets to protect, the way to extract integrity evidence and how to deliver them to the Verifier.

Unpredictability. Software attestation exploits continuous verification of the target, that is, the target is periodically attested, and several attestation responses are produced over time. Such repetitive behaviour may expose the system to unwanted vulnerabilities. Indeed, a high number of similar data generated by the protection enable an attacker to collect information and to infer patterns that may reveal internal detail of the protection scheme. The designed detection technique has to make procedures unpredictable to avoid such threats.

Unpredictable evidence extraction. It is needed to ensure that different attestations performed on the same asset generate different integrity evidence, thus avoiding that an attacker could produce fake attestation responses by reusing collected valid responses. Moreover, a technique that can protect multiple assets at the same time should ensure an *unpredictable asset selection*. In other words, integrity evidence should be extracted from a different asset for each attestation transaction, and the asset to attest should be selected randomly.

Low information exposure. Software attestation intrinsically exposes data to a potential attacker. Indeed, the client-server architecture needs to exchange network messages and to embed data into the delivered target. Given that exposures are not entirely avoidable, the design of a software attestation method has to limit such

exposures as much as possible. Hence, messages to exchange and data to embed into the target have to be designed to be *concise* in order to expose fewer data. As a side but important effect, compact data structures also let to reduce the overhead introduced by the technique. Moreover, messages and data computed on the target platform and sent over the network should be designed to be *unintelligible* from an external observer.

Diversification. Protections that are always delivered in the same way and the same form let attackers have much more chance to inspect, understand and disable it once for any target. Hence, it is necessary that the design of a detection technique let it be implemented in different versions in term of internal details. In other words, software attestation procedures must be delivered with different implementations. In this way, an attacker that defeat the protection can gain advantages just from the version he has. Moreover, a periodic change may be designed to increase the security level; that is, the protection procedure released with the target may be new ones.

Composition. Software protections are known to be weak if applied alone; then, it is usual to combine multiple protections to achieve a better security level. Hence, software attestation should be designed by taking into account the chance to combine it with other techniques. It means that all the procedures and data structures involved in the designed software attestation system must take care of other protections. Moreover, as explained before, techniques such as diversification may change the software structure at runtime. The software parts involved in the change may eventually be part of the asset that attestation is monitoring. Hence, the compatibility with other protection has to be a feature also for runtime. If compatibility is considered at design time, any additional protection technique can be applied to the target regardless of the transformation it makes on the source or binary application code. Then, if all the protections are applied before software attestation, the only requirement to avoid compatibility issues is: protections must be able to track the transformations they port. Indeed, software attestation has to be able to retrieve data from assets as they have not been changed.

3.2 Static software attestation

This Section presents the work performed on static software attestation. The main effort in realising a static software attestation system has been spent in design. That means, collecting solutions from literature, analysing them, underline their limitations and define a set of requirements for a model that can be practically used to obtain a protection detection technique for real-world applications. Finally, to test the effectiveness of the technique, an implementation of the defined model has

been delivered. The overall aim of the here presented work about static software attestation is to assess the technique from a security point of view and to evaluate its practical applicability.

3.2.1 Design

This work presents an integrity monitoring mechanism that aims at lively checking static properties of running software. In particular, the proposed software attestation system can assess the integrity of the code section of a target program that is running, thus loaded in memory. The proposed system has been designed to be modular in order to make it easily extensible and customizable.

With respect to the reference architecture, the fundamental features that a static software attestation system should support have been defined. In this way, it is possible to clearly define and semantically separate the basic tasks that are required to obtain the desired static attestation mechanism, that means, each group of basic tasks gathers a set of operations that have the same purpose in performing the attestation. Besides, it enables to design an utterly abstract system, i.e. a parametric system, that can accept different implementations for the same semantic basic tasks. In order to describe the fundamental tasks, it is necessary to specify the instantiation of the Attester and the Verifier features that, as described in Chapter 2, strongly depend on the nature of integrity evidence and on the procedures needed to extract and verify them.

Assets and integrity evidences

In order to define the procedure to extract integrity evidence, it is necessary to identify which are the software assets that the static software attestation system will aim at monitoring. Generally speaking, the detection system discussed in this section was intended to detect modifications on the code portion of a running program. In particular, it is necessary to define which is the level of detail the detection has to identify for the protected asset.

To this purpose, it is worth considering the possible granularity levels in code layout at which it is possible to establish the integrity checking: instructions, branch-less code blocks, procedures (functions), whole code section. Single instruction integrity monitoring is quite hard to achieve. Indeed, it would require to identify and to locate every single instruction at runtime. Furthermore, a single instruction does not represent an asset to protect from a semantic point of view, that means, a code asset is something that brings meaningful information about sensitive algorithms or data. For this reason, a single instruction is not considered a valuable asset in terms of security and protection.

The whole code section would be the most effective asset to protect, given that

it brings the most comprehensive information about the program features. Unfortunately, this level of granularity cannot be exploited to monitor the integrity of a program. Indeed, the whole code binary section could be extensive and monitoring it at runtime could cause a significant execution overhead. Moreover, it has not been chosen as a valid asset because it can change at runtime due to other techniques applied to the binary. For instance, software attestation could be put aside to dynamic code change techniques that can modify the loaded code directly in memory, e.g. code mobility that loads freshly downloaded code portions. In this case, the static integrity evaluation of the whole code section would be much harder and more costly than effective.

Therefore, the asset to monitor has been identified at functions level. It ensures an acceptable level of runtime overhead and to be compatible with dynamic software changes. In addition, function code blocks have enough semantics to be considered for protection purposes and to represent a valuable asset in terms of information to protect. Branch-less code blocks are useful to represent functions' disposition in memory but may lack in semantic. Indeed, they represent just a portion of a function; hence, a single block may not enclose the whole asset that is meant to be protected.

Finally, the assets intended to be protected are functions, and their representation has been designed as a set of branch-less code blocks. The advantage that comes from this decision is represented by the possibility to use the designed static detection system even if binary code manipulation techniques are applied, e.g. binary obfuscation or layout randomisation. In practice, if a binary obfuscation technique modifies the layout of the compiled program by breaking down and scrambling the code blocks to make harder to reconstruct the original control flow, it would still be possible to represent functions' information for software attestation. A function that gets scrambled is firstly divided into code sub-portions, and then, all the code chunks are shuffled inside the program and linked by inserting branch instructions that make the execution to flow from one block to the subsequent one. Then, a function can be represented as a set of blocks that execute in a specific order, i.e. a sorted list of branch-less code blocks.

Once the assets have been identified, the integrity evidence and the way to collect it from the target are the next elements to define. First of all, a representation of the assets was needed to allow the attester to recognise them at runtime. Given the nature of the monitored code regions and their possible fragmentation around the global code memory, each function is represented by a data structure, named *memory area*, which contains a list of blocks. Each element in the list stores the virtual address of the beginning of the block and its length. The beginning address is an offset from the starting virtual address of the whole code segment of the target. In addition, each memory area item that is included in the Attester data structure contains a label that uniquely identifies it for the monitored target application. This label is used at attestation time to identify the memory area on

which integrity evidence is extracted.

Given such data representation, the integrity evidence is obtained by hashing the memory area selected for attestation. However, merely hashing a memory area as it makes the attestation mechanism to be subject to trivial replay attacks. For instance, an attacker that can detect the attestation procedure may exploit it to compute valid hashes for all the memory areas and use them to reply to any attestation request, thus circumventing the integrity check. Then, it is clear that the bare hash of memory areas is not a secure measure of its integrity.

Evidences collection and verification

In order to make the integrity evidence more effective from a security point of view, it is needed a mechanism to extract evidence from memory areas that can make the evidence tightly coupled with the request that triggered their extraction. Moreover, the mechanism has to be random. Hence, it has been decided to involve the incoming nonce so that the evidence collection is driven by it.

Then, a procedure to extract evidence from the target that can be driven by a random input (the nonce) was needed. In other words, it is desirable that evidence data would be collected randomly. Moreover, the asset to attest selection has to be random to make the procedure as less predictable as possible.

Hence, it has been decided to implement a *random walk* procedure that drives the data collection from a target asset. A random walk is a pseudo-random exploration procedure of a sorted and indexed set of elements. The implemented random walk aims at extracting bytes from the selected memory area in a completely unpredictable way. To this purpose, the delivered solution provides two different random walk implementations to let the attestation system customisation to choose among them for the final protection instantiation.

Normal random walk. This flavour of random walk exploits the properties of group theory, in particular of *cyclic groups*[1]. The procedure treats the memory area to attest like a cyclic group of integers whose elements are the bytes' indexes in the memory area buffer. An assumption enabled the design of the random walk: the procedure does not apply to the whole memory area buffer but just on a portion of it, whose size is a prime number p. In particular, the selected portion's size is equal to the greatest prime number less than or equal to the area size N. This assumption lets any positive number a, strictly less than p, to be a generator for the cyclic group. In fact, from group theory properties:

$$a \in \mathbb{Z}_n | gcd(a,n) = 1 \iff \langle a \rangle = (\mathbb{Z}_n, +)$$

[1] From algebra's group theory: "*A cyclic group is a group that can be generated by a single element*".

In other words, given any group $\mathbb{Z}_n$, any element $a \in \mathbb{Z}_n$ that is coprime to the group size n, is a generator of the group under addition. Moreover, form primes properties

$$\forall p \in \mathbb{P}, \forall a \in \mathbb{Z} | a < p \implies gcd(a,p) = 1$$

then

$$\forall p \in \mathbb{P}, \forall a \in \mathbb{Z}_p | \langle a \rangle = (\mathbb{Z}_p, +)$$

In practice, the greatest common divisor of any integer number less than a prime number and the prime number itself is equal to one (by definition). Hence, a group with a size equal to a prime number is generated by any element of that group. Then, any element in $\mathbb{Z}_p$ generate all the others in exactly p steps according to:

$$\mathbb{Z}_p = \bigcup_{i \in \mathbb{Z}_p} \{(i \cdot a) \mod p\}$$

Then, by applying these considerations, it is possible to ensure that the random walk can retrieve p bytes from the considered memory area buffer with a certain level of randomness and within the fewest possible steps, which is exactly equal to p. Summing up, the random walk needs a set of parameters to work:

- *actual buffer size* p, the size of the considered portion of the memory area to attest, e.g. the largest prime number in the interval $[0, N]$;
- *generator* a, the randomly selected number, less than the actual buffer size, which will be used to generate the indexes of all the bytes to retrieve from the memory area buffer;
- *initial offset* o, the offset in the memory area buffer from which the actual buffer starts, i.e. a number in the interval $[0, N - p]$;
- *buffer size* n, the total number of bytes to retrieve from the actual buffer;

The role of all the parameters should be clear from the previous considerations, except for the buffer size. The buffer size has been introduced to increase the randomness of the collected evidence. In particular, if two requests give the same generator, offset and actual buffer size, the number of bytes to retrieve should make the hash of the collected buffers different. Indeed, this parameter introduces $n - p$ duplicated byte values or $p - n$ missing byte values in the collected buffer that do not give any additional information but alter the final hash value. All these data are deduced from nonce, then a high level of randomness is expected. Indeed, the same memory area would generate twice the same hash value with a very low probability. The probability of obtaining the same attestation response data from the two different attestation requests for the same memory area depends only on the parameters of the random walk. Indeed, any hash function gives the same result

if and only if the input data are the same. Hence, that probability is given by the function:

$$f(n,p,o,a) = \frac{1}{|p| \times |o| \times |n| \times |a|}$$

where $|p|$, $|o|$, $|n|$, $|a|$ are the number of values that can be assumed, respectively, by the parameter p, o, n, a, i.e. the size of their domain. For a fixed memory area, the value of the parameter p is always the same; hence, $|p|$ is equal to one. Given that p is supposed to be close to N, it is possible to assume $p \approx N$ for any value of N. Consequently, o becomes always null, and its domain size is equal to one. The parameter a can assume any value less than p, thus less than N. Hence, $|a|$ can be assumed to be equal to N. Formally:

$$p \approx N \implies |p| = 1$$

$$p \approx N \implies o = 0 \implies |o| = 1$$

$$p \approx N \implies |a| = N$$

Then, the probability of obtaining the same attestation response from two different attestation requests for the same memory area is given by:

$$f(N,n) = \frac{1}{N \times |n|}$$

The parameter n can vary from 0 to 2^{32}; then the probability function depends only on the size of the attested memory area:

$$f(N) = \frac{1}{2^{32} \times N}$$

For instance, for a memory area whose size is 1 KiB (1×2^{10} B), the probability to obtain two attestation responses that are the same is:

$$Pr = \frac{1}{2^{32} \times 2^{10}} = 2.27 \times 10^{-13}$$

Starting from index o, at each step i, the random walk extracts a byte from the memory area M and inserts it into the *prepared data buffer*. In the end, the prepared data buffer will be hashed to obtain the integrity evidence to be sent to the verifier. Then, the i-th byte of the prepared data buffer is obtained according to:

$$B[i] = M[o + ai \mod p]$$

where B is the prepared data buffer and M is the byte array of the memory area to attest.

Goldbach random walk. This second version of the random walk extends the previous one. Instead of working on the whole selected buffer from the memory area

to attest, the procedure is applied to two sub-buffers. The Goldbach decomposition[2] gives the sub-buffer partition. The actual buffer size is odd since it is a prime, then it gets decremented to obtain an even number. The couple of primes whose sum gives that buffer size value is searched in a pre-computed and hard-coded set of Goldbach's couples. Hence, two sub-buffer sizes p_1 and p_2 are obtained, and the data will be extracted from these two buffers.

The actual buffer size could be made equal to the highest even number less than or equal to the memory area size, However, it has been decided to implement the same method to reduce the considered memory buffer in order to keep the randomness level close to the one obtained for the normal random walk. In other words, if the random walk considered the whole memory area, the initial offset parameter would become useless, and the randomness introduced by this parameter would be lost.

The Goldbach random walk produces a buffer of n bytes extracted from a given memory area in n steps. At each step, the procedure extracts one byte from both the partitions, it adds together the two extracted bytes and inserts the resulting byte in the attestation data buffer. In practice, once the two partition buffers have been identified, the normal random walk procedure is applied separately on each of them. Hence, it is necessary to have two generators, one for each partition. The generators are obtained from the generator a selected for the original actual buffer size, in the following way:

$$a_1 = a \mod p_1$$

$$a_2 = a \mod p_2$$

In the end, the Goldbach random walk needs the following parameters to extract the evidence from a memory area with a size of N.

- *Actual buffer size* p, the size of the actual part of the memory area to attest, i.e. the highest prime number in the interval $[0, N]$, from which the two sub-portions sizes form a Goldbach partition of $p - 1$.
- *Generator* a, the randomly selected number that would be the generator for $\mathbb{Z}_p$ from which the two partitions' generators are obtained, as it happens for the normal random walk
- *Initial offset* o, the offset in the memory area buffer as for the normal random walk, i.e. a number in the interval $[0, N - p - 1]$;

[2]The Goldbach conjecture states that every even number greater than four can be expressed as the sum of two prime numbers. Hence, given an even number n greater than four, the Goldbach decomposition gives, as a result, a couple of primes, p_1 and p_2, whose sum is equal to the number n: $p_1 + p_2 = n$.

- *Buffer size* n, the total number of bytes to retrieve from the actual buffer.

At the i-th step, the Goldbach random walk produces one byte in the final buffer according to:

$$B[i] = M[o + ia_1 \mod p_1] + M[o + p_1 + ia_2 \mod p_2]$$

The considerations of the probability of collisions in attestation responses made for the normal random walk are valid also for the Goldbach random walk. Notice that this random walk version does not introduce any practical advantage compared to the normal random walk; hence, they provide the same security level. Nevertheless, it is reported to demonstrate the capability of the overall attestation mechanism to be flexible, extensible and to be compliant to the diversification and composition requirement.

Parameters As seen in the previous sections, the data preparation procedures need a set of parameters to work correctly. As anticipated, the randomness of the process and then the predictability of the produced integrity evidence depends on the input parameters. The central role in producing random parameters is played by the random nonce sent with the attestation request.

A subset of the parameters that are deduced from the nonce have already been presented in the random walk Sections, hereafter the full list:

- *area label* m is the ID of the memory area to attest among the monitored ones;
- *buffer size* n, the total number of bytes to extract from the memory area to generate integrity evidence;
- *actual buffer size* p, the size of the considered memory area portion;
- *generator* a, the generator number for the random walk procedure;
- *initial offset* o, the displacement from the beginning of the memory area where the actual buffer starts.

All this data is obtained by manipulating the incoming nonce value and represent a preliminary set of parameters. The deployed procedures may additionally manipulate the decoded parameters in order to adapt them to the needs. Given the modularity of the designed system, it is possible to change the way these parameters are calculated from the nonce. It permits to diversify the possible instantiation of the protection and to them extend them. The delivered work includes four different implementations of the parameters extraction from nonce that are presented in Section 3.2.3.

As emerges from the previous discussion, the nonce is used as a sort of random seed to obtain all the parameters and values needed for attestation purposes. The nonce is generated using a cryptographically secure pseudorandom function from the OpenSSL library named `RAND_bytes`. The fixed scheme to retrieve information allows the nonce to be manipulated, at generation time, to drive the attestation on the client in an arbitrary way. It can be a useful feature but may come with a drawback. If the randomness of the sent nonces is altered, the attestation mechanism gets more vulnerable to attacks like replay or cloning. Then, it is preferable to avoid that the distribution of the nonce values is altered to avoid any help to attackers.

3.2.2 Pre-computing attestations: the Extractor

The Extractor is a bare functional component as it is not foreseen by the reference architecture presented in Chapter 2. It has been introduced for the proposed static software attestation method to reduce the verification time and the computational load at run-time. Indeed, it takes care of some tasks that are required by the Verifier, but that can be performed offline before the attestation process is actually in place.

The Extractor pre-computes and stores couples of nonces and associated valid attestation data. Practically, it randomly generates a set of nonces and, from an untampered version of the protected target, it computes the attestation data. That is, it performs the proper random walk on the memory areas, calculate the hash using the proper function and stores the resulting digest on the database. Then, the Verifier runtime tasks are eased and speeded up as it has just to compare the received digest with the pre-computed one.

However, the Extractor is very efficient; it can produce hundreds of nonces and related attestation data in seconds on an off-the-shelf laptop. Moreover, the nonce generation can be easily parallelised. The Extractor is a helper component that is neither exposed as a service nor directly reachable by clients. It is only invoked by the Manager when the available pre-computed data are about to finish.

3.2.3 Application of static software attestation

Once the protection mechanism has been designed, the next step was to bring the protection to be usable in real life. To this purpose, it has been designed a toolchain to protect a given application automatically. Figure 3.1 reports the workflow of the protection toolchain. A developer who wants to protect his application with static software attestation has to provide the source code with proper annotations.

Annotations

The annotations are provided together with the source code through a custom directive: `#annotation(<params>)`. The user can specify two different types of annotations: Attester declaration and memory area definition.

Attester declaration annotation. This annotation declares a Attester and defines its parameters. As described in Section 3.2, a Attester can be configured through a set of parameters that describe the technique's key features. Consequently, this annotation characterises also the Attester related components, i.e. Actual Verifier and the Extractor.

The Attester declaration annotation can be inserted in any place in the application source code. The first parameter to pass to the annotation directive defines the type of annotation, namely `declaration`, and states that this annotation describes a Attester to insert into the target application. The declaration directive takes a set of parameters in turn:

- `RW` parameter, it specifies the kind of random walk that has to be used to extract data from monitored code regions, as described in Section 3.2.1. The possible values are:

  `RW_NORMAL`, the Attester will implement the normal random walk;

  `RW_GOLDBACH`, the Attester will implement the random walk based on Goldbach's hypothesis.

- `HF` (Hash Function) parameter, it selects which hash function the Attester will use to generate attestation digests, that is which function must be applied to the prepared data to obtain the attestation response. It allows five different values:

  `HF_BLAKE2`, it selects the Blake2 hash function;

  `HF_MD5`, it selects the MD5 hash function;

  `HF_SHA1`, it selects the SHA1 hash function;

  `HF_SHA256`, it selects the SHA256 hash function;

  `HF_RIPEMD160`, it selects the RIPEMD160 hash function.

- `NI` (nonce interpretation) parameter, it specifies how nonces are interpreted to extract parameters for the random walk. It allows four different values, `NI_1 ... NI_4`, each them specifies a different way to obtain all the parameters for the selected random walk: the selected code region to attest, the buffer size, the actual buffer size, the random walk generator and the initial offset. For the sake of simplicity, the details of each `NI` version are not reported here as they do not provide any additional information from a research point of view.

- `NG` (nonce generation) parameter, it specifies the way the nonces are generated. Only one implementation for nonce generation is provided. It generates the nonces randomly and does not allows further customisations. For this reason, only one value is foreseen, by now, for this parameter: `NG_1`.
- `MA` (memory areas) parameter, it specifies the memory area management API to use. It selects the way to represent memory areas internally. In the current implementation, memory areas are a list of sorted branch-less code blocks; in other cases, it may be different. The API interface abstracts the access to monitored memory areas as they have not undergone any transformation, as they are linear and contiguous memory segments. By now, just one value is accepted for this parameter, namely `MA_1`.
- `DS` (data structure) parameter, it specifies the data management API to use. This API is used to parse attestation requests and prepare the attestation response, to read and write requests and response components, to manage attestation prepared data and hashed data. Just one implementation has been delivered for this parameter; then it accepts just one value: `DS_1`.

The declaration annotation parameters are meant to specify the single primary group of features implementations that have to be used to generate an Attester. Given that the architecture was designed to be modular and to work independently from the actual implementation of its components, the value allowed for each parameter can be extended to accept other implementations. It means that if it is needed to tailor the software attestation system for a particular kind of hardware or low-level software architectures, it is possible to generate ad-hoc attestation components that fit the system features. If a new implementation of any fundamental task will be defined, it must provide the relative source file and the relative parameter value. The file to provide must implement, respectively:

- `RW`: `ra_data_preparation.h`;
- `HF`: `ra_do_hash.h`;
- `NI`: `ra_nonce_interpretation.h`;
- `NG`: `ra_nonce_generation.h`;
- `MA`: `ra_memory.h`;
- `DS`: `ra_data_table.h`.

Other than this group of parameters, the `declaration` annotation requires a label specification and a frequency to associate to the Attester that is being declared. The label is specified by using the `label` parameter and identifies the Attester among the others in the application; it has the following format:

```
label(<name>)
```

where <name> is a string specified without any quote. The frequency parameter has this form:

```
frequency(<seconds>)
```

where <seconds> is an integer value that defines the number of seconds between two subsequent attestation requests to be sent to defined Attester. Finally, here it is an example of declaration annotation:

```
annotation(
    declaration(RW_NORMAL, HF_SHA256,
                NI_1, NG_1, MA_1, DS_1),
    label(att_n1),
    frequency(100)
)
```

This annotation requires the inclusion of a Attester, named `first`. The server must be configured to query this Attester once every 100 seconds, on average (`frequency(100)`). The Attester uses the SHA256 hash function (`HF_SHA256`), performs the normal random walk (`RW_NORMAL`) to prepare attestation data and interprets received nonces according to nonce interpretation version 1 (`NI_1`).

Memory areas definition annotation. The second kind of annotations for software attestation is the `region`. This annotation is used to select a function whose code has to be to monitored by software attestation. Then, it has to be put before a function definition.

The main parameter to pass to the annotation statement is the `region` parameter, it specifies the annotation type. There are two required parameters for this annotation. The first specifies the reference to the Attester for the memory area. The second one tells whether the area has to be attested at the target start-up or not. The Attester reference is specified using

```
associate_to(<label>)
```

where <label> is a non-quoted string of characters. If the specified label is not present in the list of defined Attesters the annotation interpretation process fails. The attest-at-startup parameter has the following form:

```
attest_at_startup(<bool>)
```

where <bool> is a non-quoted string that can be either `true` or `false`. This parameter specifies the need to send an attestation request for the selected code region as soon as the Attester connects to the Manager. The attest-at-startup parameter is optional; if omitted, it is equivalent to specify false as its parameter. An example of a memory area definition annotation is:

```
#annotation(
    region,
```

```
    associated_to(att_n1),
    attest_at_startup(true)
)
void foo(int a, int b){
  /*...*/
}
```

This annotation specifies that the code of function `foo` has to be protected with static software attestation, the Attester that is in charge of monitoring this area is the one with label `first` and this function have to be mandatory attested when the target connects to the server.

Annotations are extracted by a Perl script, the *Annotations Extractor*, which generates a JSON file that lists all the encountered annotations. A specialised tool processes the annotations in the JSON file: *Annotations Interpreter*. The Annotation Interpreter is a Java-written tool; it receives the annotation and creates a folder for each Attester declaration. Each generated folder contains two files: a descriptor and a frequency file. The descriptor file, named as the Attester label, specifies the reference to implementation files of the attestation modules for that Attester (as requested by the annotation). This file contains a set of variable definitions following the syntax used by makefiles. Each variable definition specifies the name of the real implementation file (.c) of the associated fundamental block. The content of this file is, for instance:

```
RA_DATA_PREPARATION_BLOCK_NAME := ra_data_preparation
RA_DO_HASH_BLOCK_NAME := ra_do_hash_sha256
RA_NONCE_INTERPRETATION_BLOCK_NAME := ra_nonce_interpr_3
RA_NONCE_GENERATION_BLOCK_NAME := ra_nonce_generation
RA_DATA_TABLE_BLOCK_NAME := ra_data_table
RA_MEMORY_BLOCK_NAME := ra_memory
```

The second file, the frequency file, contains the required attestation frequency, as it is interesting only for the server-side logic. The second file is named `<label>.freq`.

Besides, the interpreter checks that each defined Attester has at least one memory region associated with it. In practice, defined Attesters that do not monitor any memory area are ignored. Moreover, the interpreter checks the unicity of the Attesters labels.

Building steps

After the annotations extraction, the toolchain proceeds by building each Attester, then by bundling all the Attesters together. The *Attester Builder* is the tool in charge of pursuing this task. All the implementation files are identified and selected from a descriptor file. The entire source code contains placeholders to prevent symbol names conflicts in the final build due to duplicated Attester files that contain the same symbols. Then, a preprocessing phase is needed: each selected

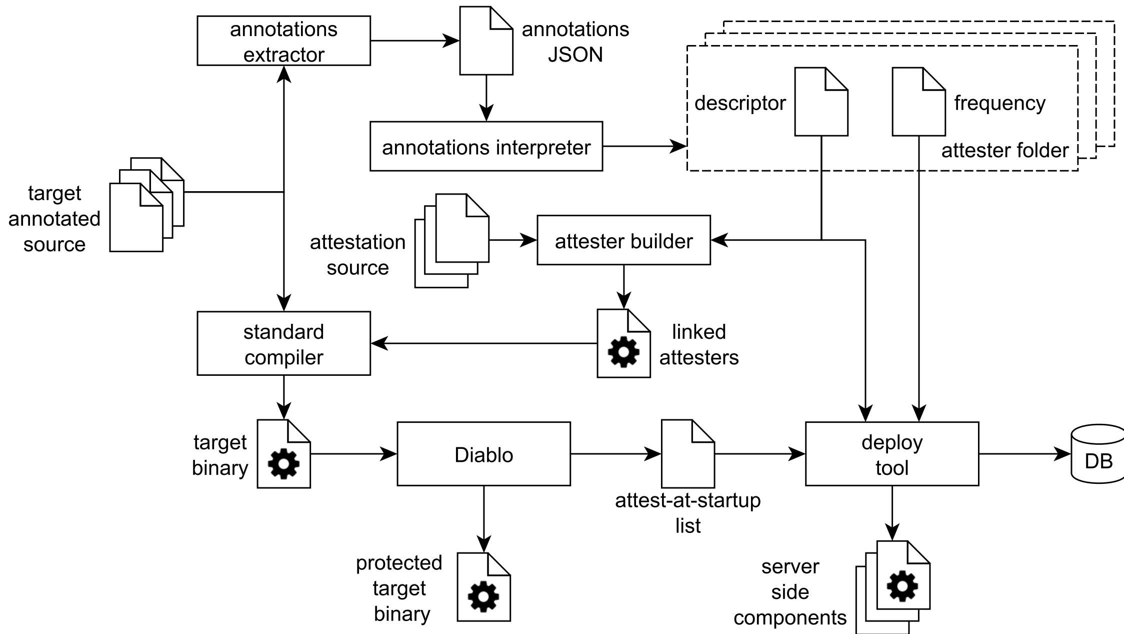

Figure 3.1: Automatic application of static software attestation workflow.

block source file is processed, and the placeholders are replaced. For example, the function that prepares attestation data is declared as

```
RA_RESULT ra_prepare_data_NAYjDD3l2sEzY7Xcs(RA_table t)
```

the `NAYjDD3l2sEzY7Xcs` part of the function name is replaced with the relative Attester name, which will be unique for the program. The placeholder string is the same for every global symbol in all the source files. This way, all the global symbols introduced by attestation files are surely made unique for the linking phase. In addition, the nature of the placeholders let the files to be compiled even without any replacement, that is useful for server-side components compilation. The Actual Verifier and the Extractor are compiled according to the Attester specifications. Indeed, Extractor and Actual Verifier relate to only one Attester; thus, they do not suffer any naming conflict as they include one and only one implementation of every remote attestation implementation file. Then, after the preprocessing phase, the name of the previously presented function would become, for instance:

```
RA_RESULT ra_prepare_data_label00001(RA_table t)
```

All the characterised files generated by the preprocessing phase are collected into a folder for each Attester. In addition, that output folder will contain all the non-parametric source files that are needed to build the relative Attester, i.e. those files that do not need any customisation. At the end of this step, there is one folder for each declared Attester, which contains all the source files ready to be built to generate an Attester object.

Afterwards, the toolchain invokes the compiler (that can be chosen by means of configuration files) in each Attester folder, thus obtaining an object file for each Attester. All the Attester object files are then linked together to produce a unique object file that contains all the Attesters. On their tuns, the Attesters object file is linked together with the protected application.

Since the static software attestation system has been developed in the context of the ASPIRE project, some steps of the toolchain depends on third-party tools. Indeed, the memory layout of the final binary strictly depends on binary transformations that can be applied, for instance, by a layout randomisation tool. In the project context, Diablo[3] has been the chosen layout randomiser. Diablo acts like a linker that transforms the given binary into another one that has a different structure but the same behaviour and semantic. Then, the randomiser is asked to generate the data structure that describes the final memory areas' layouts, to pack it into a binary portion, named *Attestation Data Structure (ADS)*, and to link it together with the target application. Hence, Diablo is in charge of extracting information about code blocks, labelling the memory areas and reporting areas that have to be attested at start-up. The randomiser lists the memory areas that were flagged with "attest-at-startup" in a file for each Attester[4].

After the completion of the randomiser tasks, the toolchain builds the server side elements and creates the records in the database (Section 2.7) for the newly protected application, through the *Deploy Tool*. Hence, the deploy tool starts the final setup stages. The descriptor file generated from annotation processing defines the building blocks also for the Actual Verifier and the Extractor. Then, the deploy tool runs the standard compiler to build properly characterised files and to deliver server-side components as well. For every new application that gets protected, the software attestation toolchain inserts a new line in the application table, with a freshly generated application ID. Then, for every Attester inserted into the application, one line in the Attester table is generated. Moreover, the deploy tool launches the extractor to compute a set of nonces and related valid attestation data. The amount of precomputed data can be customised throughout an input parameter. After that, the deploy proceeds by processing the attest-at-startup list file for each generated Attester to record all the specified memory areas in the database.

[3]https://diablo.elis.ugent.be

[4]An acknowledgement is due to thank **Bert Abrath** for the work performed to extend Diablo, which made the tool work together with the static software attestation mechanism presented in this document.

3.2.4 Security analysis

The proposed architecture comes with a set of drawbacks that can be generalised to any other static software attestation system. Considering the MATE scenario, where the end-user and the attacker have the same privileges on the delivered application, it is possible to underline the vulnerabilities that one have to take into account when static software attestation is deployed. An attacker has to interrupt the Attester's tasks or to fake its behaviour to prevent the protection to work correctly.

Attestation operations prevention

Hereafter are presented a set of possible threats that are introduced by implementation choices made for the Attester. It generally means that most of the presented issues may be overcome with enough engineering effort and do not require any further research.

Preventing the Attester to start, the most trivial attack consists in avoiding the execution of the Attester. Indeed, if the Attester does not run at all, it cannot start any communication with the server; thus, the Manager would not be aware whether the target application is running or not. This kind of attack is possible, for the presented system, because the Attester run in a thread that is parallel to the target application's main process. An attacker that can identify the point in which the Attester thread is spawned can bypass it as well, e.g. by statically removing the system call to thread spawn of dynamically bypassing it.

Countermeasure: engineering is needed to make the Attester routine integrated into the target business logic, thus to be not distinguishable from the regular application tasks. Additional instrumentation operations should be introduced to rewrite the target's code so that it includes commands to start communications with the server and reaction to incoming requests.

Requests Stop, another attack that can be mounted is to stop incoming requests. Only by monitoring the system calls that receive data from the network, an attacker can catch the communication and drop it before it reaches the Attester logic. It is noticeable that it is not equivalent to disconnect all the application from the network. Indeed, a full disconnection would prevent network business logic operations as well. For instance, for a real-time streaming application network disconnection would cause the playback to stop as the stream is not reachable any more.

Data Collection Inhibition, even if the attestation request reaches the attester routine, it is possible from an attacker's point of view, to avoid the execution of attestation collection procedure. Indeed, once the attestation request is received, the Attester thread calls the data preparation routine that is a function. Hence, an attacker may identify the function call and, statically or utilising a debugger,

she can force the execution to pass over that code, thus not running it at all. This way, the attestation response is not computed and not sent to the Verifier.

Attestation Response Send Stop, an attacker can tackle the attestation process even after the attestation data is computed and response is produced. It is possible to prevent the Attester from sending responses back to the server both from within the Attester process and from an operating system perspective. In the first case, the attacker identifies the invocation to the network send system call and bypasses it, e.g. by using a debugger. In the second case, the attacker captures the traffic from the target to the server and drops packets from outside the application, e.g. by using a packet filter, a firewall or a network traffic monitoring tool.

Countermeasures: the way to face all the previous three vulnerabilities strictly depends on the kind of reaction that is put in place. If reactions are only local, which means that the server does not trigger faults in the application, the threat is more significant. That is, local reactions are usually based on events that can be observed locally, e.g. timers that are reset by a positive event. Hence, an attacker that can prevent requests to reach the Attester should be able to bypass this kind of reactions as well. On the other hand, if reactions are server-based, the vulnerability is much more negligible. The server-side mechanism can notice that no responses are received against sent requests. In this case, an attacker cannot do anything to block the reaction as the server can, for instance, stop to serve the application with the requested contents. Consequently, countermeasures are more straightforward if reaction enforcement is performed remotely, by the server, and they become more laborious if the reaction enforces locally.

Cloning attacks

The vulnerability presented here affects any static attestation (both hardware and software). Hence, it affects the delivered system as well. Van Oorschot *et al.* presented this kind of attacks and demonstrated that it is possible to circumvent self-hashing-based techniques for integrity protection in a subtle way [51]. The presented attack exploits on operating system modifications; it exploits the difference between memory accesses to data and code.

The attacker clones the untampered target program, tampers with one copy, and let the other one intact. Given that data memory accesses can be easily told from code-fetch memory accesses, the attacker can modify the operating system kernel so that it redirects all the data accesses (including those to code portions performed by an Attester) arbitrarily. Then, the attacker runs both application copies and let the patched kernel redirect all the data reads to the genuine copy in memory while letting the instructions to be fetched from the tampered program. Hence, when a self-hashing mechanism tries to read from the application code, just like the Attester does, it gives valid data even if the running application is corrupted.

Finally, the attack is enabled by the MATE scenario and fully privileged access to the execution environment. The attacker can perform any tampering on the target structure, which is the kind of attacks that static software attestation aims at detecting. Moreover, the attacker must not reverse engineer the application to spot protection mechanisms such as software attestation; he can approach the application as a black-box and concentrate on the sensitive parts to modify on his needs.

Countermeasures: unfortunately, Cloning Attack is a severe issue for static software attestation in general and the mechanism presented in this Chapter. There is no remediation to this kind of attack, as it is still an open issue in literature as well. The attack cannot be spotted from within the application because it is performed at a higher level, that is, at the operating system level. Then, there is no chance to block Cloning Attacks by using a technique that is entirely based on target embedded components. The proposed architecture for static software attestation offers possible mitigation to the cloning issue. Diversification and runtime renewability may be exploited to replace the embedded Attester(s) periodically. Consequently, it would periodically replace hard-coded procedures that extract evidence as well. It would cause computed attestation evidence to differ from the ones obtained by the original Attester even if the attacker redirected memory accesses. Nonetheless, run-time renewability techniques are out of the scope of this work; hence, this document will not discuss them.

Non structural modification

Another class of vulnerabilities is represented by tampering modifications that do not alter the application structure, those modifications that alter the program behaviour without tampering with its binary code, e.g. Debugging Attacks. An attacker can use whichever external tool to stop, restart and redirect the application execution, as well as to modify the content of data locations (memory, registers, data segments). In practice, these kinds of attack dynamically alter the target's execution while letting the software structure intact.

Countermeasure: this kind of attack cannot be detected by static software attestation by definition. Indeed, static software attestation aims at discovering structural modifications to the original program and is not conceived to protect from this kind of threats. To this purpose, it is worth to evaluate the possibility to combine static software attestation with additional protections, e.g. dynamic software attestation.

In conclusion, the static software attestation system presented in this chapter is in line with most of the others found in the literature. For instance, it checksums code portions to attest the structural integrity of the target and exploits random nonce-based transactions. On the other hand, here it has been presented a static

tamper detection system that can be extended, customised and diversified to exploit different procedures and assets kinds, as opposed to many works in literature. Indeed, immutable structures and procedures represented a vulnerability for many static attestation systems, as reported in [51]. Hence, flexibility features allow the proposed mechanism to be tailored to resist to attacks that defeat the delivered Attesters by offering the chance to vary the Attesters themselves. In particular, it enables diversification as each fundamental part of the system can be extended and rewritten. Finally, it enables composability with other protections to enhance the security level of the resulting system and to protect the detection mechanism itself.

3.3 Dynamic software attestation

The work performed in the context of dynamic software attestation was about obtaining an attestation system to reason about the robustness of the security model, to investigate the effectiveness of dynamic features for attestation purposes and to assess practical applicability of such system against real-world attacks. As anticipated during the background presentation (Section 1), dynamic software features' integrity seems slightly harder to be attested than the static one. Indeed, dynamic attacks are subtler and harder to spot because they do interfere with program behaviour and with features that are not related to their structure. Then, it is not enough to monitor something that the application is made of; instead, it must monitor something that the application does. For instance, the code reported in Figure 3.2 naively checks the validity of an input key against a secret. The code uses the function `validate` to validate the input key and, if that fails, the program quits. As it can be deduced from the source code, it is confirmed by the disassembled code: if an attacker can force the `validate` function to return 1, whatever the input key is, the check process gets bypassed. In practice, an attacker could attach a debugger (e.g. `gdb`) at the beginning of the `validate` function, make it to not execute at all and return 1 (e.g. `return` command provided by `gdb`[5]). As demonstrated, it is not so hard to modify the behaviour of a program without altering its structure. Hence, this is the kind of attacks that dynamic software attestation tries to detect.

The main challenge in this scenario is then to represent software behaviour with a rigorous model that can be easily attested. In practice, a given behavioural feature has to be translated into a formal representation in order to obtain a logic or mathematical expression that can be verified by an automatic process.

Chapter 1 presented several attempts to pursue representation of a software

[5]GDB documentation: Returning from a Function:
https://sourceware.org/gdb/current/onlinedocs/gdb/Returning.html#Returning

```
char secret[256] = secret_str;

int validate(char secret[], char key[]){

  for(int i = 0; i < 256; i++)
    if(!secret[i] ^ key[i])
      return 0;

  return 1;
}


int check(char key[]){

  if(!validate(secret, key))
    exit(EXIT_FAILURE);

  return 1;
}
```

Figure 3.2: Source code that checks a key against a secret.

application from a semantic point of view. One among the others exploits the so-called software *Invariants*. Invariants are logic predicates that are expected to be true during the whole execution of a program or just a portion of it. For instance, with reference to Figure 3.2, from function `validate` it is possible to infer `i > 0` `i <= 256` as a couple of statements that are always evaluated as true. An attacker that tries to alter the behaviour of the program in order to bypass the checking loop in the `validate` function may modify the value of the variable `i` at the beginning of the loop by giving it a value greater than 256. It will cause the loop not to execute even once, thus making the function always returns 1. This kind of attack is expected to alter the invariant `i < 256`, then a mechanism that monitors that invariant is supposed to recognise that attack.

Historically, invariants have been introduced to describe programs' constraints, e.g. for assertions and programming by contract [23, 28, 31], or to detect programming errors and incorrect implementations at runtime [17, 26]. Indeed, one of the first applications of invariants appeared in software engineering literature, to detect bugs before they take place at execution time [28]. Afterwards, invariants have been exploited to define axioms and constraints that describe software correct execution by deriving them from specifications as demonstrated by Gries [23]. In addition, several works proposed to exploit invariants to identify and locate software faults and problems [14, 9, 10, 33, 19]. As these former software engineering works stated, invariants were manually defined starting from program specifications and then checked against the implementation. All the early usage of invariants were

related to software execution, thus behavioural, issues. It suggested employing invariants as an integrity metric to evaluate execution correctness as they create relations among software runtime features, such as variables.

In early stages, invariant assertions were manually defined. This approach is not very effective; indeed, it has been as demonstrated that user-specified invariants are likely to be absent in the final application [20]. Then, it is not possible to rely on assertions that the designer or the developer of software expects to be valid from specifications. This issue led to the introduction of dynamically-inferred likely-invariants. *Likely-invariants* are quite the opposite of classical "true" invariants, they are deduced by analysing execution traces. These traces are collected by executing the application against a set of (considered valid) inputs [20]. Dynamically inferred invariants are then not unquestionably valid and can fail, by chance, even if the application is sound. Hence, likely-invariants are empirically deduced statements that are valid during the application execution (or a portion of it) with a certain level of confidence.

Consequently, there have appeared several works presenting automatic inference tools for likely-invariants deduction, such as IODINE [27] and Agitator [11] from industry, and DIDUCE [26], Dysy [18] and Daikon[6] from academia. Daikon is the most popular tool for likely-invariants inference, it is developed and maintained by the University of Washington, and it is released as a free and open source tool.

Likely-invariants have been used as an integrity measure, for code or data, by several works in literature. Lorenzolli *et al.* used likely-invariants to recognise failure contexts [37]. They inferred invariants from these contexts then, if the invariants are valid, their system can state that the target environment is about to enter in a failing state. As a countermeasure, whenever a failure is detected, the protection mechanism tries to correct the execution issue or to bypass the execution of the failing part. ClearView is another example from Perkins *et al.*; it analyses a running system and, using Daikon, it describes its behaviour in terms of likely-invariants [41]. In case of failure, ClearView can automatically detect invariants that are no more valid and to propose a patch to restore their validity. In such a way, ClearView can protect from code-injection attacks. ReDAS is a software attestation mechanism able to monitor dynamic system properties, such as global variables. Daikon infers likely-invariants at system call invocations, and then, they claim to detect tampering modifications at runtime by evaluating these invariants [32]. Gibraltar is a solution based on likely-invariants that aim at detecting kernel level rootkits. An inference phase deduces likely-invariants from data structures and internal values of the Linux kernel, such as entropy pool, processes list, page sizes. The detection phase continuously evaluates the inferred invariants in order

[6] Daikon is involved in a number of publications that can be found at https://plse.cs.washington.edu/daikon/pubs/

to detect tampering attacks [7]. Wei *et al.* exploited "scoped invariants", likely-invariants that are valid for a limited scope in a program, to detect execution anomalies [55].

All this background led to invest in studying likely-invariants as an execution correctness evidence, thus in studying likely-invariants based attestation, namely *Invariants Monitoring*. The primary purpose was to check the effectiveness of invariants in describing software behaviour and to detect dynamic attacks on a running software application. In other words, the next sections describe the instantiation of the abstract software attestation model considering likely-invariants as evidence, hence, studying the capability of such evidence in modelling the software execution. More generally, the work about dynamic software attestation exploits likely-invariants as a specific modelling tool to assess literature claims, verify that the attestation model can manage dynamic software properties and try (if possible) to extend the drawn conclusions to all the dynamic software attestation systems.

3.3.1 Invariants Monitoring

Invariants monitoring is a software attestation technique that aims at verifying the execution correctness of a target application by checking likely-invariants inferred on its dynamic features. The main idea is that likely-invariants may be a valid tool to define constraints and limitation to the possible executions states of a program, then, to spot outlier conditions that may be introduced by an attack.

In terms of assets to protect, likely-invariants seemed a valid metric to measure the integrity of two different, but related, software aspects. First, likely-invariants can be used as evidence of data integrity. Indeed, statements given by invariants are relations that limit the program variable data domains. It is not possible to assume that a variable is valid if it is in a specific range, but, on the other hand, it is possible to state that something happened if it exceeds that range. Second, likely-invariants can be a metric for execution correctness. Logic assertions given by invariants can be interpreted as pre-conditions to execute a piece of code, or as post-conditions after the piece of code is executed. As for data, an application cannot be assumed to be sound if these conditions are satisfied in the place they were expected to stand. Anyhow, unmet condition statements can indicate something wrong in the program execution. These considerations represented the reason that finally pushed this work to investigate the Invariant Monitoring technique.

Problem statement: protection solutions presented in literature exploit likely-invariants as an execution correctness measurement. Hence, they assume that there exists a direct link between data constraints, i.e. likely-invariants, and software behaviour that allows such software properties to be considered as dynamic integrity evidence. Unfortunately, for what concerns the author, literature does not present any work that formally assesses this relationship.

Hence, the remainder of this chapter presents the design of a likely-invariants-based monitoring system for execution correctness and validate the model assumptions on which the technique relies. The research novelty brought by this work is then represented by the discussion about the likely-invariants expressiveness to the purpose of security-grade software behavioural modelling.

Given that such an assessment is quite hard to be performed from a theoretical perspective, dynamic software attestation has been analysed from an experimental point of view in order to grasp empirical results. To this purpose, practical implementation has been developed to:

- automatically inferring invariants on a program;
- model allowed program behaviours using inferred invariants;
- retrieve data from running program to verify invariants, thus to check execution correctness and valid behaviours.

3.3.2 Design

This work aims at fitting the lack of an attestation mechanism able to lively checking dynamic software properties on running programs. In particular, the proposed software attestation system can retrieve data from a running system and evaluate the execution correctness of it through previously inferred likely-invariants.

Invariants inference and modelling

The first step is then represented by the inference and selection of invariants that could model the target application. To this purpose, the design included a preliminary training phase. During this phase, the target software is instrumented and run under Daikon.

It is clear that Daikon plays a crucial role in this attestation system. Daikon deduces and checks likely-invariants against observed runtime values of the target execution traces. Daikon has a front-end instrumentation tool for C programs, namely *Kvasir*. In order to run the application and collect traces, the incoming source code has to be compiled with debugging symbols and without any optimisations. Then, the built application runs under the control of Kvasir. At the end of the execution, traces are collected and can be passed to Daikon for the inference analysis. The inferred likely-invariants are statistically filtered out in order to minimise false positives. In practice, Daikon discards those statements whose probability of being a mere coincidence is over a certain threshold, the ones that pass this check are elected to be valid likely-invariants.

Daikon, as the other inference tools, is designed for software engineering purpose, then it may introduce some issues when used for security purposes. It can

check for invariants at functions entry and exit point. It can be a feature for maintenance goals, but it limits the usability for security purposes. The attestation mechanism has to be able to check the target at any moment, not only in precise program points, because attestation requests may come arbitrarily during the application execution. Indeed, the validity of likely-invariants is statistically sound just in the points where they were inferred, elsewhere they may fail by chance, e.g. an involved variable may not be consistent with the others in the considered invariant. On the other hand, it is not possible neither to define nor to infer invariants that are valid at any point of a program. That is because of the program's data tightly depend on the point where it has been evaluated. It is the first drawback that comes from likely-invariants. This limitation is related to another drawback that has to be addressed. Most of the attacks are mounted internally to a function, at code blocks level. Hence, invariants that do not describe the code inside functions, but just at the beginning or the end of them, are useless for detection purposes.

For this reason, an attempt has been made to extend the invariant points of detection. Daikon documentation suggests a trick to make the tool to infer invariants from loop variables: by adding a call to a dummy function that takes the variables that have to be considered for invariants as input parameters. Based on this idea, a code manipulation tool has been developed. The tool aims at injecting dummy functions in every inner scope of the C-language syntax. It means that every syntax statement that generates a new scope, and potentially introduces new variables, will be modified so that all the internal variables are passed as parameters to a dummy function. Hence, the *Functions Injector* is an ad hoc tool, made in Java by exploiting ANTLR[7] libraries to explore the Abstract Syntax Tree. For each new scope that is encountered during the exploration of the Abstract Syntax Tree, the injector lists all the continued variables, generates a function with an empty body that takes all the variables as input parameters, insert the function definition before the last explored function and insert a call to the just generated function at the exit of the scope. To prevent dummy function from being removed by the compiler, their body is not left empty; instead, it filled with a simple `printf` call that prints all the input variables. It should ensure that the dummy function will be in the compiled program. That is, it should enable Kvasir to trace all the local variables then Daikon to infer invariants on inner scope variables. For instance, for the code of the following function:

[7] https://www.antlr.org

```
double f(int a, int b){
  int c; double d;
  c=a+b;
  d=(double)a/c;
  return d;
}
```

the injector lists the `c` and `d` variables, then, generates the function:

```
void _____injectedFunction_rand (int c, double d){
  printf("%x %x\n", c, d);
}
```

where `rand` is a 32-characters random string added to make the function name unique for the processed program. Finally, it injects a call in the original function and the function definition:

```
void _____injectedFunction_rand (int c, double d){
  printf("%x %x\n", c, d);
}

double f(int a, int b){
  int c; double d;
  c=a+b;
  d=(double)a/c;
  _____injectedFunction_rand (c,d);
  return d;
}
```

At this point, the inference task can be performed to model the target's behaviour. All the likely-invariants obtained by this process represent the way to prove target integrity. The evidence that the Attester would report are the actual values of target data that permits to evaluate invariants, i.e. variables and data structures of the program. In practice, the Attester extracts data involved in likely-invariants from the target and send it back to the Verifier. The Verifier puts proper values into likely-invariants, evaluates the relative logic statements and decree target application soundness.

Check process

After the design of the integrity evidence and proving method, the next crucial point to address was to define the way to collect data from the running target. The likely-invariants inferred by Daikon mainly involve variables, arrays and structures (`struct`). The remainder of this section uses variables to refers to all these different kinds of data. It is needed by the Attester to be able to retrieve this kind of data from the target, while it is running. In practice, the Attester have to know the location and size of interesting program data for any execution point of the program.

To this purpose, the procedure lists all the variables involved in all the selected inferred invariants. The list of interesting data groups variables into *global* and *local* variables. Local variables are grouped by the function they belong to, and global ones are all grouped for the entire program.

Variables can eventually be in any possible memory place: stack, data segments, register, known memory location, memory location referenced by a register or translated as a constant value. In addition, the location of a variable can change during the execution and can be different depending on the instruction pointer value. Then, the Attester needs hints to identify variables' location depending on the program state. To this purpose, the procedure exploits DWARF[8] debugging symbols. These symbols are additional information that a compiler inserts into the built executable binary to ease debugging operations. Among the other information, DWARF symbols associate the location of a variable to an instruction pointer range, which specifies the code segment for which the location is valid. DWARF symbols depend on the compiler (e.g. gcc or llvm) and, of course, the information associated with symbols depend on the target platform architecture (e.g. x86 or ARM).

At the instrumenting time, the procedure compiles the target application with DWARF symbols; afterwards, it identifies the interesting variables and extracts the related location information from debugging symbols. This process leads to populate a data structure that records all the variables that the Attester has to extract, their location description and uniquely identify them. Thus, a *Variables Data Structure (VDS)* is generated and will be made available to the Attester at runtime.

At runtime, the Attester tries to collect as many interesting variables as possible and send their values to the verifier. The variables' values extraction is best efforts made. Indeed, it is improbable that all the variables are available for any execution point in which the Attester can stop the program. Hence, the Attester scans the VDS, and if it expects variable to be available for the execution point reached by execution, it retrieves that variable value. If a variable does not exist at the moment of retrieval, it will be not included in the evidence buffer.

As said, the variable availability depends on execution points. During its working, the target may call an arbitrary number of functions in a nested way, i.e. a function calls other functions, thus generating a call stack. Hence, the valid execution points are all the ones present in the call stack. Then, the Attester has to be able to unwind the call stack starting from the last function encountered at the program stop to the first function invocation.

Finally, when attestation is needed, the Attester stops the application and starts

[8]DWARF is a debugging symbols standard produced by the DWARF Standards Committee available at http://dwarfstd.org/.

the variable extraction as follows:

1. identify the current execution point, i.e. instruction pointer, and the relative stack frame;
2. retrieve values from the locations specified in VDS for that execution point;
3. move upwards in the call stack, if the function called is in the program address space, restart the process from point 1, otherwise finishes.

Once all the available values have been retrieved, the attester builds the data to send to the verifier according to:

$$d = n||(v_{ID}(v_i), \text{Value}(v_i))||\ldots||(v_{ID}(v_i), \text{Value}(v_i))$$

where n is a 16 bit integer that counts the total number of collected variables, $v_{ID}(v_i)$ is the unique identifier of the variable v_i and Value(v_i) is the value of the variable v_i found in memory. The Attester builds the final data to send to the Verifier as follows:

$$r = d||H(d||N||ID|S)$$

where H is a hash function of choice (SHA1, SHA256, and Blake2 are supported by now), ID is the unique identifier the running application, and S are data that relate the client to the server (e.g. secrets shared during the mutual authentication).

The central database stores the inferred likely-invariants in a form that eases the verification phase. In particular, they are logical expressions that state relations between variables. Daikon reports likely-invariants using variables' names that relate them to a function. Then, the procedure modifies the expression before its storage: it replaces each variable name with a placeholder that specifies the identifier of the variable to use to evaluate the invariant.

Hence, when the Verifier receives the response data r, it first checks the correctness of the digest by computing the hash against data d. Then, Verifier scans all the likely-invariants associated with the application in the database, selects those for which all the involved variables are present in received data and tries to evaluate them. For each retrieved likely-invariant, the Verifier replace all the variable placeholders with the proper value and the obtained expression is evaluated. Finally, the Verifier stored the result in the central database.

3.3.3 Application of Invariants Monitoring

In order to bring the proposed attestation mechanism to a real-life usable level, it has been developed a (nearly) automatic procedure to apply protection to a given vanilla application.

The overall input of the process is the set of target's source files that has to be protected. From a high-level perspective, the automatic application of dynamic

software attestation follows two parallel and synergic sub-processes: *discovery of likely-invariants* and *extraction of DWARF information*.

The first sub-process instruments the given application to make it ready for Daikon processing and finally collects likely-invariants. This sub-process outputs the list of invariants to check at runtime and the list the variables to locate and retrieve from the running target. The second sub-process is in charge of collecting all the information that would allow the Attester to collect variables values at runtime. The sub-process obtains a build of the application containing the DWARF debugging symbols. Then, it scans the obtained binary: the DWARF symbols are parsed to collect all the data about interesting functions and variables life-cycle. Moreover, the sub-process identifies and labels each variable that the procedure will be monitoring. Finally, this sub-process produces the VDS blob and makes it available for the Attester. Hence, the obtained VDS binary file will be injected into the final protected binary.

In the end, the two sub-processes merge into a common final step. An interpreter processes the extracted likely-invariants. It relates the involved variables to the invariants expressions and stores the resulting statements in the central database according to a format that is suitable for verification.

Figure 3.3 provides a closer look into the protection workflow. Hereafter, a fully detailed description of the single components is reported.

Standard Compiler. This component is the standard compiler of choice that is used to generate executable binaries, e.g. gcc. The choice of which compiler to use is up to the final user. The only requirement for this element is that it has to be compliant with DWARF debugging symbols so that it can produce proper binaries containing proper symbols. This component is the only one whose execution cannot always be made automatic. Real life applications usually come with a build script (e.g. Makefile) that contains the proper commands to build it. Given that build procedures must be compliant to Daikon's requirements, it will be necessary to manually modify building scripts by the user (e.g. removing optimisations or enable debugging options). Then, for the steps that involve the Standard Compiler, it may be required manual user intervention.

Function Injector. This component manipulates the source code of the original application to protect. For each inner scope found, it lists all the declared variables and generates an ad hoc function that takes those variables as input parameters (as described in Section 3.3.1). The output of this component is then an altered version of the source code and a description file. The description file reports, for each function in the original program, all the injected functions. This output allows the next components in the workflow to keep track of the injected functions, i.e. to go back to the original function, starting from a known injected function.

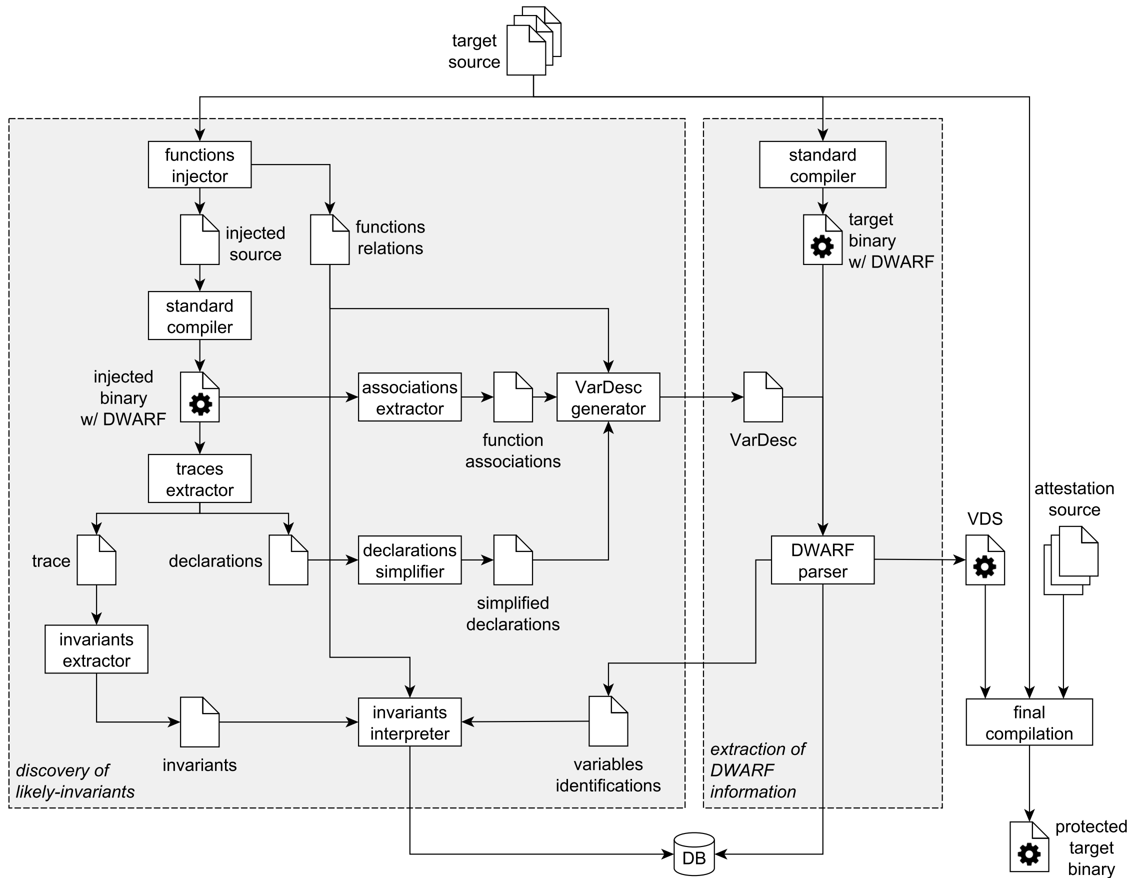

Figure 3.3: Automatic application of Invariants Monitoring workflow.

Traces Extractor. This component executes the application containing injected functions. Then, before this component is invoked the output source code from Function Injector is compiled using the Standard Compiler. Given that the invariants inference tool selected for this prototype is Daikon, it is mandatory to use its front-end instrumenter to collect traces, namely Kvasir. Kvasir requires that the application is compiled with debugging flag enabled (e.g. `-g` for GCC) and without any optimisation (e.g. `-O0` for GCC). The Traces Extractor has to deliver, as an outcome, a trace that is compatible with Daikon expected input format and a declaration file. The declaration file reports only the program's variables that are interesting for invariants, thus ignoring all the others.

Invariants Extractor. This component properly invokes Daikon by giving it the collected execution traces. The inferred invariants are listed and delivered to the next steps.

Associations Extractor. This component is in charge of listing all the functions that are present in the binary, built after the function injection phase. Each function is associated with its compilation unit full path name. This component prevents problems that could come from duplicated filenames in the folder structure. Indeed, DWARF symbols identify functions with respect to compilation units, i.e. source files.

Declaration Simplifier. This component elaborates the declaration file produced by Trace Extractor. It selects the name of variables that the technique has to be able to extract from the protected running application. In practice, it removes syntax terms that are specific by merely leaving the bare name of the variables for each found function.

VarDesc Generator. This component combines the data from the Function Injector, the Associations Extractor and the Declaration Simplifier. This component aims at generating a file, namely the *VarDesc* file, to specify which are the information to extract from the binary built from the unprotected sources. It will output a properly formatted file that specifies, for each compilation unit, the functions and the dependent variables for which locations and life-cycles have to be extracted by the Attester at runtime.

DWARF Parser. Starting from the VardDesc file, this component analyses the DWARF information contained in the unprotected application binary. To this purpose, the original source code is preliminarily built by using the Standard Compiler with debugging flags enabled (e.g. `-g` and `-g-dwarf` for GCC). The DWARF Parser produces the VDS file, enumerates all the variables by assigning them to a unique identifier and record them into the central database along with the data about its owner function and compilation unit. Identifications of variables are also output in a file for the next steps in the workflow.

Invariants Interpreter. This component elaborates the invariants extracted by Invariants Extractor and rewrites them according to the variables identifications. In practice, it replaces the variables names with the respective unique ID so that each invariant expression is directly associated with the involved variables according to a custom syntax. For instance, the following invariant:

```
_low + _limit <= 100
```

would become

```
@{127} + @{619} <= 100
```

by assuming that database IDs of variable `_low` and `_limit` are respectively 127 and 619. The interpreter stores the obtained expressions in the database for the verification phase.

Final Compilation. In this step, the toolchain generates the final protected target. It builds the unprotected source code along with the attester one and rewrites the obtained binary to inject VDS into it. The resulting application is the protected target application that is ready to be delivered.

3.3.4 Security analysis

Once the prototype has been developed and tested, a set of experimental evaluations have been performed on it. These evaluations aimed at assessing the validity and the applicability of Invariants Monitoring. Moreover, it aimed at identifying the possible limitations and security issues that could arise. A set of five applications have been set up for protection as real-life use cases:

- *Lynx*, a command line web browser[9], used to simulate Man-in-the-Browser scenarios;
- *MOC (Music On Console)*[10] and *mpg123*[11], two command line music players used to simulate a DRM scenario;
- *oathtool*, an OTP generator and verifier[12] used to check Invariants Monitoring effectiveness in high-sensitivity authentication software;
- *gamespace*, a test application for ncurses library[13] that is properly dimensioned and interesting enough for an empirical assessment;
- *Bzip2*, a command line compression tool[14] that is representative for a large set of utilities that are useful for daily tasks.

Hereafter, a subset of these use cases is discussed as they are representative for all the others and the same conclusions can be extended to all of them. Each discussed use case is reported to underline different considerations and conclusions that are nonetheless valid in general. Hence, the following discussion will report one aspect for each discussed use case just for the sake of conciseness, but all the drawn considerations are generally valid for all the analyses applications.

Table 3.1 reports statistical information about each considered use case application. All the invariants inferred on the use cases have been published[15].

[9] http://lynx.browser.org
[10] http://moc.daper.net
[11] http://www.mpg123.org
[12] http://www.nongnu.org/oath-toolkit/oathtool.1.html
[13] https://sourceforge.net/projects/game
[14] http://www.bzip.org
[15] https://github.com/vucinic/im

The MOC use case: protection process and attacks assessment

The application allows the user to play and pauses songs, organise songs in playlists that can be played linearly or randomly. The original MOC code has been modified in order to simulate a DRM system. Hence, the original MOC application plays the role of *Premium* version, with fully enabled features. The modified copy acts as a *Free* version, and it restricts the access to some features: it only allows the user to play playlists in random order and it does not allow the user to skip tracks and arbitrary navigation within playlist songs.

The ported changes have been implemented as `if` statements that enable functionalities based on a pre-compiler define:

```
if( PREMIUM ){
  /* perform original task */
} else {
  /* write a message */
}
```

The define and the unused branch of the statement are not present in the final binary due to compiler optimisations. Then, the modifications remove the functionality and make the two versions of the application different. The changes prevent the execution of the *next, previous* and *toggle shuffle* commands at the user interface level. The user interface takes the selected command from the user and forwards it to the proper player functionality. In addition, the player has been modified (for the *Free* version) in order to make the jump to the next song a random skip, i.e. prevent to play a playlist sequentially.

Afterwards, the protection procedure has been deployed. Sensitive functions to protect have been selected[16] to be monitored. Then, the toolchain has been applied to protect the application.

Invariants inference issues have been encountered while the protection was applied. In the face of small execution traces, i.e. 100 MB, protection is applied in 5 h. With 3 GB execution traces, the toolchain took 50 h, but it has been stopped when the 50% of the trace was processed. As a last trial, 1 GB execution traces led to 140 h tool chain execution and 1% processed traces. By observing this evidence, traces size could not be the only parameter that affects Daikon elaboration. Maybe, also the type of collected traces affects the performances. Unfortunately, this conjecture could not be proven because performances widely varied over executions against the same set of input; thus, the consideration is drawn by experience.

Once the execution extraction process has been tuned to run within an acceptable time, Daikon has discovered 8553 invariants. Manual analysis on the inferred

[16]Namely, the functions are the `main` in `main.c`, `go_to_another_file`, `audio_play`, and `audio_queue_move` in `audio.c`, `go_file`, `play_it`, `cmd_next`, `menu_key`, and `options_get_int` in `interface.c`

invariants revealed that most of them were redundant as repeated both as function pre- and post-conditions or because they were related only to global variables whose value remains the same for the whole program execution. Finally, 246 distinct invariants were identified to be interesting but related to global variables. The traces extraction process has been further tuned to ignore global variables, then 24 interesting invariants have been found.

Function injector revealed to be ineffective. Unexpectedly, the tool does not provide any additional information to improve Daikons invariants inference. MOC exploits an intensive usage of global variables, then a limited amount of additional invariants were expected due to Function Injector but, surprisingly, no invariants came out. The analysis looked for causes of this behaviour in the developed code, extracted traces and Daikon process. The deployed toolchain executes on the use case application. The collected traces contains references to the injected functions and the dependent variables. Unfortunately, Daikon does not infer any invariant on the additional functions. Then, the work performed suggest that the responsibility for this issue resides in Daikon internal implementations, which are out of the scope of this work. The study observed this behaviour also for the other use cases.

The assessment analysis tested the attack detection ability of the technique. A user wanting to use *Premium* features on the *Free* version of the modified MOC application well models an attack in the analysed context. Two different attacks have been deployed.

1. Disable of *shuffle mode* to enable arbitrary order in playlist reproduction. This attack performed by using a debugger to alter the content of `Shuffle` option in the `options` global variable that is evaluated by `go_to_another_file` function and bypassing the code that randomises the playlist order.

2. Enable the *next* function to allow the user to skip songs in a playlist. This attack also involved the debugger, it traps the call to `go_to_another_file` function and executes the previously removed code to jump to next track in the playlist.

Each attack underwent 100 attestation transactions. The first attack gets always detected because it alters a variable that is involved in a monitored invariant; thus, the detection success rate reached 100%. For the second attack, 100% of false negatives has encountered. Indeed, the attack does not alter any variable value; thus, no invariants are affected, just because it is out of the technique's possibilities. Other use cases have confirmed the analysis performed on the MOC player use case.

The MOC use case has been used to evaluate the performance of the Invariants Monitoring prototype. At client-side, Invariants Monitoring uses resources to compute the attestation response. The attestation response's computation time is the sum of the time to retrieve the values of the selected variables from memory, the time to retrieve the ID of each variable from the VDS, assemble the response data, and the time to compute the hash.

Target Application	**Files**	**Functions**	**SLOC**	**Likely-invariants**
Lynx	264	1890	193625	∅
Bzip2	1	106	7010	5770 (193)
MOC	91	1215	43478	8553 (246)
mpg123	92	161	38060	7529 (3492)
oathtool	85	225	20469	3943 (2661)
gamespace	3	81	1752	29172 (18545)

Table 3.1: Statistical information on the use cases. ∅ indicates that Daikon was not able to infer invariants in a reasonable time. Likely-invariants in parentheses are the non-redundant ones.

From experimental considerations, the average time to compute an attestation was 442 ms, while the average number of variables in memory was 87 and the average attestation data length was 22 591 B. It is worth to notice that the prototype stopped the application's execution; thus, the attestation time only depends on the number and size of variables in memory and the hash algorithm speed. Therefore, the experiment evaluated the same impact on the other protected target applications. Since that target applications were expected to be attested with an average frequency of a few minutes, the delay introduced by Invariants Monitoring can be considered manageable.

At server-side, the Manager uses resources to prepare an attestation request. However, it is only the time to generate a 256 bit nonce, while Verifier has first to compute a hash then use the ID of the variable to retrieve and check the invariants. The average measured time to verify an attestation response was 1.3 s, being the average number of invariants to check 165. This result may suggest scalability issues. Nonetheless, proper engineering of invariants indexing, usage of data structures for fast verification, and, if really needed, dedicated hardware (e.g. FPGAs) may grant scalability.

The Bzip2 use case: analysis of invariants

The Bzip2 use cases enabled the discussion about likely-invariants semantic modelling properties and their meaningfulness for integrity monitoring purposes. The use case application is relatively small, and the protection process took 27 min to complete. As expected, inferred invariants were associated with precise points in the program, i.e. functions entry and exit points.

Invariants discovered from `BZ2_bsInitWerite` function were 1645, 458 of them were pre-conditions and 1187 post-conditions. Most of the invariants inferred from the function exit point do not give any additional information to those at the

entry point. Indeed, they report that a variable or a statement has not changed during the execution of the function. These kind of invariants are utterly useless for protection purpose as they do not represent any program's behaviour or feature that the implemented scheme can verify. Indeed, the Invariants Monitoring Attester stops the application as soon as an attestation request is received. When a program is interrupted, it is improbable to find the execution in a function entry or exit point, but probably right in its middle. It also means that the likely-invariants used to model software execution must be selected among the whole set of inferred pre- and post-conditions. Only those conditions that are manually recognised to apply to the whole reference function are chosen for attestation. This fact forces the user to manually filter inferred invariants to select those that could actually be useful.

Environment-specific invariants are invariants that generate from data that depends on the execution environment, e.g. file paths. For instance, Bzip2 manipulates input files and generates output files whose path is processed during the use case execution. These values pass through the program execution and generate the following invariants:

```
inName == " ../inputs/in_big.txt"
outName == " ../inputs/in_big.txt.bz2"
```

Obviously, this class of invariants is completely useless for protection purposes. Indeed, any execution would invalidate them as it cannot be assumed that input and output files have the same name and reside in the same path for any execution of the program in any possible environment in which it could run.

Consequently, a manual selection has been performed to select invariants that could be used to tell tampered application from a genuine one. Invariants can be classified, and their properties can be discussed.

From Bzip2 documentation, the `workFactor` property is a value that can be specified as an input parameter, and it is saved into a global integer variable when the execution starts. This parameter specifies how high repetitive input is managed and determines when to force the usage of fallback algorithms. The default value for the parameter is 30 that gives an acceptable behaviour over a wide range of cases. During the toolchain execution for protecting the Bzip2 application, the value of this parameter has not changed, even over multiple executions. Hence, the inferred invariants report the value of `workFactor` variable always equal to 30:

```
workFactor == 30
```

Consequently, legitimate users that specify, for instance, the `--exponential` flag to select the compression algorithm would make the `workFactor` value not equal to 30 any more. Hence, a legitimated user would be considered as an attacker, and valid executions of the application would be identified as tampered. It means that that kind of invariants is prone to produce *False Positives* because the strict dependency

on the inputs may generate invariants that are *too restrictive*. An informal consideration can be done on this behaviour and should lead to a formalisation of the problem. Let us consider the domain of an invariant that is the Cartesian product of the domain of each variable involved in it. Invariants are potentially restrictive if the measure of the part of the domain that maps to true is negligible compared to the part that maps to false.

Broad invariants are the opposite of restrictive invariants. They very loosely limit variables' admitted values, so that they are useless for modelling and integrity monitoring purposes. Broad invariants come from executions that make a variable value to assume many and different values. An example of this class of invariants is:

```
size > 1
```

In practice, broad invariants are useless because (nearly) always map to true. Although broad invariants do not introduce a real security issue as they would not generate false positives or negatives. Invariants monitoring should not consider these invariants in order to reduce memory and time consumption during the technique execution.

Inconsistent invariants come from relations among semantically unrelated variables. For instance, several invariants about arrays use independent variables as indexes:

```
BZ2_rNums[longestFileName] == 733
BZ2_rNums[longestFileName-1] == 214
BZ2_rNums[workFactor] == 419
BZ2_rNums[workFactor-1] == 472
```

This kind of invariants do not represent an issue from a security perspective but does not contribute to model the behaviour of a program or its valid states.

The invariants presented for the Bzip2 use case and the related issues have been observed also during the analysis of the other use cases. Thus, they confirm issues to be always present and diffused, regardless of the target's nature.

The oathtool use case: extraction strategies

This use case has been useful to reason about how invariants' semantic vary depending on the collected execution traces, thus about strategies for tuning invariants inference and improving their behavioural descriptiveness.

As previously discussed, false positives come from restrictive invariants due to non-exhaustive execution traces. On the other hand, broad invariants could be due to executions that let variables to vary too much. Then, the analysis reported here tried to precisely identify the causes of the restrictive and broad invariants related to the execution traces and, eventually, to suggest the best way to collect traces in order to prevent.

Oathtool can generate and validate event-based HMAC OTPs against a key and a window of computation iterations. An event-based HOTP is deterministic, as it only depends on the given inputs and the constant internal values. The experimental procedure extracted invariants from the validation phase: the application takes as input the HOTP to validate, the key and the iterations window in which the verification has to be performed and outputs verdict about the validity of the provided input HOTP.

Single execution traces. Initial analysis executed the application just once against a HOTP to validate, then inferring invariants from the obtained traces. Daikon extracted 1580 invariants, 892 of them related to global variables and 688 only related to local ones. Two main behaviours emerged from the studied invariants. First, most of the invariants are meaningless; that is, they do not describe any security or behavioural feature of the application. Moreover, most of the invariants are duplicated in different program points. Second, several invariants seem appealing at first glance, but, in the end, they revealed too restrictive. Indeed, using a single execution trace, invariants may represent a single precise case of execution that is strongly related to the input parameters. Then, the inferred invariants generated over-fitting as they are too specific and will likely be invalidated by any execution that takes different input values. For instance, the oathtool validation process stores the given input HOTP in a variable and Daikon deduce invariants like this:

```
otp == "328482"
```

It is not acceptable as integrity evidence as it is valid in only one specific case.

The situation given by a single execution trace could very likely lead to false positives. Deduced invariants are too tightly bound to a subset of the inputs and may decree perfectly valid target applications as compromised, whose only fault was to execute with different inputs.

Multiple execution traces. As an attempt to overcome the restriction that comes from limited executions, the experimental procedure considered invariants that generate from multiple executions' traces. In particular, the HOTP validation process has been run 100 times in two different ways: by giving it always the same input HOTP value to validate and by giving a different value to each execution. Traces have always been collected using valid HOTP values to be validated.

The first execution strategy (single value) led to gather 2643 invariants (1823 involving globals, 820 involving only local variables). The second attempting strategy (diversified inputs) gave 2660 invariants (1838 involving globals, 822 involving only local variables).

Therefore, it was expected to obtain more descriptive invariants and less useless invariants. However, the collected evidence has not confirmed expectations. Indeed,

the overall number of collected invariants increased, but the additional invariants are not useful and do not describe any interesting property of the application. Moreover, restrictive invariants observed in the single execution as potentially useful, turned into broad invariants (e.g. `moving_factor >= 0`) or simply disappeared (e.g. none of the invariants involve the `otp` variable).

Merging traces. An alternative to computing invariants on all the traces, it has been attempted to extract invariants from different single executions and then to merge them. In this case, several invariants were contradictory, e.g. two invariants involving the same array stated that it contained exactly one value and more than one value. This behaviour suggests ignoring all the conflicting invariants unless a manual inspection suggests a better strategy (e.g. merge them in a broad invariant).

In conclusion, the analysed target application is deterministic, and its execution is supposed to depend on the provided inputs directly. Thus, it was expected to infer useful invariants. However, traces collected from a limited number of executions produce invariants that are tightly related to the input values but do not describe the general behaviour of the application. To better describe the application, the experimental procedure used multiple execution traces with different input values. Nevertheless, this strategy makes specific invariants disappear. Instead, the extractor infers broad invariants that are useless for integrity checking purposes. Even worse, the merge of invariants collected from different executions may introduce the risk to find conflicting predicates.

The gamespace use case: empirical assessment

In order to further support the experimental analysis of the technique, data from an experiment involving master students from Politecnico di Torino have been used. The experiment asked the students to attack the *gamespace* application, a testing application for the `ncurses` library. The application presents to the user an interface where a set of entities are moving. The user is allowed to move one piece, with respect to hurdles, in the preferred direction. Each prompted direction makes the piece moving by one step in that direction. Internally, the application uses a data structure to keep track of the interface elements position and a set of functions to move pieces around the interface[17]. Each graphical element, namely a `Player`, is represented as a list of `Point` elements. Each `Point` element specifies the position of each point of the figure to draw. When an element has to move, the moving functions update the `row` and `col` fields value of all the `Point` in the list of the associated figure according to the specified direction.

[17] `movePoint`, `moveSolidPlayer`, `movePlayer`, `movePointInMap`, `moveFlexiblePlayer`

Participants were asked to perform an attack that consisted of forcing the application to move of two steps at each keypress in each specified direction instead of one. The participants were asked to write down a small report to describe the action performed to port the attack and, possibly, to attach the modified source files.

The collected reports underlined that successful attacks exploited a minimal set of application changes. Then, the study analysed the ported changes in order to understand if Invariants Monitoring could be able to detect them.

The involved students mainly followed two statistically balanced strategies: modify data and modify control flow. For the first strategy, they modified the content of `point->row` and `point->col` before the moving functions were invoked. This kind of attack is supposed to be identifiable by Invariants Monitoring; in fact, it alters variables' values. However, it was not detected. A further investigation reported that it is due to broad invariants. The attacks is ported by doubling the movement, that is doubling the increment of the `point->row` and `point->col` variables. The attacker takes care of keeping the movement in the allowed area in order to not crash the application. Unfortunately, invariants that could be violated by these changes only restrict the domain of those variables to the movement area. Given that the value of the two variables is always in the allowed range, the attack violated no invariants. Indeed, the `point->col` and `point->row` variables keeps to be valid with respect to invariants even if they are altered.

The only way to identify this kind of attack would be to compare the position of points recorded from two subsequent executions of one of the moving functions on the same point. However, currently, stateful information is never used to build invariants but considering more dynamic, and stateful information in invariants could be exciting future work.

As a second way of attack, students doubled the calls to the moving functions without any change to variables (both global and local), thus implementing a code replication attack or a debugging attack. This kind of attacks is not even supposed to be identified by the technique, as confirmed by evidence.

Program size and inferred invariants. From Table 3.1 it is worth to notice the strange relationship between the number of inferred invariants and program size (SLOC) introduced by the gamespace use case. Indeed, from other use cases, the SLOC/invariants ratio is equal to 4.2 on the average, and in the gamespace use case, it becomes equal to 0.6. It means that the amount of inferred invariants is nearly 15 times the SLOC value, this completely reverses the previously observed trend that sees the number of inferred invariant to be (at least) less than the SLOC value. The explanation for this behaviour must be sought in the software structure of the use case.

Gamespace involves a high number of global variables that are, besides, involved in many functions calls that are, in turn, very frequent. Hence, the number of

inferred invariants becomes very high (compared to the program SLOC) due to the high number of possible combination of variables and operations at each program entry and exit point. Finally, this behaviour suggests that it is not possible to deduce any relationship between program size and the number of invariants that generate from it.

Execution issues

Along with issues that come from the usage of invariants as a modelling tool for application behaviour, the protection execution disclosed a set of practical issues.

As aforementioned, inferred invariants are pre- post-conditions associated with functions. Hence, they can only be evaluated when the associated functions execute, even for invariants only based on global variables. The selected use case applications model common behaviours in real life programs. Indeed, those programs are in the idle state most of the time, while waiting for (user) input. During the idle period, they execute very few functions are, often just the main (e.g., the library functions that play tracks), thus few variables are available in memory. Consequently, there exists the risk that either the protection evaluates almost always the same invariants or checks no invariants at all if the idle functions are not monitored.

Similarly, when the Attester stops the program, all the invoked functions are not in their entry or exit point. Hence, the invariants inferred for those functions may be slightly invalid because the point in the program in which they are evaluated is not precisely the same as the one in which they were inferred. Furthermore, invariants introduce an issue that is directly related to the moment in which variable values are extracted for evaluation. That is, the Attester must extract values of variables involved in the same invariant from memory at the same moment; otherwise, the invariant itself may be invalidated. It is intrinsically due to likely-invariants nature, likely-invariants are deduced from variables' values from a precise execution point at a precise execution moment. Then, if variables are extracted from different executions or different execution instants in time, they may not be consistent. For instance, given the following code snippet:

```
for (i=0;i <100; i++) {
  j=100 -i;
  /* do something with i and j */
}
```

An invariant that could be deduced is `i+j==0` but, at verification time, if values of variables `i` and `j`

In addition, the developed system is a kind of best efforts approach. The Attester tries to retrieve all the variables that are somewhere available in memory among the monitored ones (i.e. those in VDS). It could cause consistent memory load and consistent amount of data to include in attestation response. Indeed, this is very noticeable when most of the monitored variables are global (i.e. always

available, thus always included in attestation response) and when the variable size is large (e.g. big data buffer). This issue introduces a network overhead and a computation overhead for the Attester routine. The overhead is linear with the number of monitored variables and their size. Even worse, the higher the overhead becomes, the easier to identify, locate and stop the Attester routine is, just because the execution of the Attester becomes longer. Thus, an attacker may notice that execution remains in a suspect point of the application for an extended period. Another issue comes from practical needs. The delivered Attester has to access the same memory area as the monitored application. Hence, Attester cannot be an external entity and has to be embedded into the target application at most as a thread.

Invariants Monitoring limitations

Given all the considerations reported in the previous sections, it is clear that the protection technique is faulty and has to be improved to be practically usable. To resume all the issues and limitation that the protection exposes, a classification has been made based on two main classes. Limitations that are Specific of invariants (S) are issues intrinsic to the use of invariants. In this class, limitations related to the use of invariants for software protection are explicitly marked (with P). On the other hand, technological limitations (T) that depend on tools, procedures and implementations choices to extract invariants and use them for remote monitoring purposes. Every limitation is associated to a severity level, in the high (h), medium (m), low (l) range. The severity level expresses how hard it should be to overcome the limitation to the purpose of using Invariants Monitoring as real software protection. In this way, it is possible to classify limitations in the compact form ($classes; severity$). According to this class model, the full set of identified limitation for Invariants Monitoring is reported hereafter. Moreover, the discussion will propose mitigations (where possible) for each of the presented limitations.

Concurrent extraction ($S, T; m$) of variables' values causes invariants to be evaluated as invalid due to time-inconsistency for extracted variable values. It could be mitigated by considering data dependency information. It seems that some invariants could be verified with values from different executions while others will only be evaluable with values taken at the same time, regardless of the technological improvements. Further research is needed to overcome this issue.

Pre- and post-conditions ($T; m$), invariants are inferred only at begin and end of functions. This limitation could be overcome by inferring Invariants within functions' body, e.g. by injecting code (as presented for the inner scope) or waiting for new tools to do it. However, the impact of this improvement needs an estimation. It may only be an improvement for large functions when variables have longer and more complex life-cycles.

Inter-function invariants $(S, T; l)$ are not considered, that is, invariants are inferred as statements that relates variables from the same scope. It can be mitigated with further research that also considers data dependency. Indeed, evaluating invariants with variables from different functions could represent an improvement for the quality of the verification.

Variables' types $(T; l)$ represent an issue that is tightly related to Daikon, which can consider only global variables, functions' parameters, no data structures and no inner scopes. This issue is important but not severe because the number of invariants that can be found is already enough to model crucial software aspects. However, overcoming this issue could allow Invariants Monitoring to better model the target behaviour.

Impossibility to infer invariants on arrays $(S, T; m)$, it is also an issue related to the tool of choice, i.e. Daikon. To overcome this issue could be not so easy. Indeed, despite the technological evolution, arrays and matrices may contain data that are difficult to correlate and may increase false positives. It some cases, it could be useful to instrument the application to flatten multi-dimensional arrays. More investigation would serve to assess this limitation better.

Sending of all the variables $(T; m)$ involved in invariants may be problematic for bandwidth consumption, further research is needed. Attestation requests that explicitly ask for a set of variables have been considered but discarded. Indeed, all variables but the global ones are only available in memory when a specific part of the code executes. Thus, the Attester is legitimate to answer that a variable is not in memory, and an attacker may legally bypass the protection by always answering that no variables are available in memory. A version of the Attester that only sends specific variables may mitigate this issue. For instance, the Attester could send only the variables whose value changed since the last attestation or just a subset of the global variables, and it would require implementation effort to obtain complex and state-aware Attesters. Furthermore, this solution must be investigated to identify the real potential improved that it may bring in.

Constructor option $(T; l)$ is used to let the Attester to start and initialise itself before the actual target starts. It lets the Attester to be spot and removed very quickly. The impact of this issue is shallow since it is only related to the implementations presented in this document and, given that the system delivered to this purpose is a research prototype, it should be easy to overcome with engineering and implementation effort.

Non-discoverable attacks $(P; h)$, unfortunately, there is nothing to do when attacks can be mounted by attaching a debugger without altering variables values. Indeed, it is out of the scope of this protection technique that is not able to detect attacks that do not alter variables values. Formerly, when the study of the technique begun, it was expected that Invariants Monitoring should be able to detect any dynamic attacks by merely assuming that invariants were an excellent tool to model all the aspects of the software behaviour. Unfortunately, the practical

experience underlined that there is no chance to detect attacks that do not alter variable values, e.g. attacks that redirect the execution flow.

False positives $(S; m)$ take place whenever an invariant is invalidate by the collected evidence, but the target is still sound. This issue can be reduced by increasing the traces used to infer invariants. Theoretically, with full coverage, no false positive should be found because invariant would become more precise. However, the number of likely-invariants could decrease.

The most severe and challenging to overcome issue is related to *false negatives* $(P; h)$. Attacks that pass unnoticed are undesired for a protection technique. As demonstrated, a large set of attacks are not detected by the technique even if they were expected to be identified. False negatives are a fundamental limitation to the usage of likely-invariants for protection purposes. Indeed, they came out form all the analysed use cases. From the analysis of likely-invariants and manual inspection of a massive amount of logs produced by the Verifier of the developed system, false negatives are due to the lack of a precise link between attacks that compromise the assets in the target application and the extracted likely-invariants It invalidates the baseline assumption for using likely-invariants for protection purposes. Even if this dissertation can only discuss the experience from the analysed use cases, witnessed practice suggests that it is a general limitation of likely-invariants. Trace collection and invariants inference are statistical processes that do not consider program semantic, nature of assets, desired security properties (e.g. integrity), attack paths and strategies. In other words, it is not possible to assume that an attack that compromises one or more assets will surely also violate invariants. Hence, a valid inference among invariant violations and attacks is a research issue of crucial importance. Without such inference, it is not possible to consider Invariants Monitoring as a valid software protection technique, and it is not secure to use it in practice, to protect execution correctness for real-world applications. This limitation highlights the inability of Invariants Monitoring in detecting attacks for which it was designed, in addition to those attacks that the technique is not able to detect by definition.

Injection of the VDS (T, m) includes sensitive information about assets inside the protected program. Usually, software protections remove as much information as possible from the target (e.g. strings stripping, symbol tables removal) to reduce information provided to attackers. Then, the injection of VDS data in the delivered target weakens this practice as eases reverse engineering and secrets stealing. Indeed, the VDS can be better hidden in the application binaries (e.g. with obfuscation) but needs to be made available at the client. Enough engineering effort may suffice to overcome this limitation. Anyhow, it does not threaten the general protection model.

Manual effort $(T; l)$ is needed to compile for Kvasir and Daikon they require particular symbols and optimisation flags. It is not a significant issue and appears sustainable, yet annoying. In addition, if the inference tool is not Daikon, it is

likely to have other practical requirements on the compilation process. However, it seems feasible with better extractors or extensions of the existing ones, and with more modern standards.

3.3.5 Discussion

The study exhaustively investigated Invariants Monitoring form different points of view. Once applied to use cases, the protection method disclosed a broad set of limitations that threaten its protection strength. Improvement attempts have been performed to overcome as many limitations as possible, but the most critical issues remain open. Most of the open limitations merely need engineering and practical effort to be overcome.

Technological limitations do not affect the effectiveness of this technique significantly. Indeed, they can be addressed to make Invariants Monitoring work in practice. Techniques that avoid attaching debuggers should assist invariants Monitoring. Indeed, there is nothing to do when attacks can be mounted by attaching a debugger without altering variables values.

Two main limitations remain severe an hard to be overcome. False positives and negatives do affect the possibility of using Invariants Monitoring for protection purposes. False positives have been observed to be very likely and leading to triggered tamper countermeasures even if the target application is correct. The number of false positives may scare developers wanting to protect their applications using Invariants Monitoring because of the management costs. Indeed, the effort can be spent to collect extensive execution traces or to train a policy system to tolerate a certain threshold number of violated invariants according to their probability to fail (it will require to compute the invariants' failure probability as well). The learning phases, which should be done by developers before publishing the application, may be very time consuming and may have to be repeated at every new version of the application. Furthermore, eliminating the invariants that lead to false positive may reduce the detection ability of IM, like happened with broad invariants. Hence, this work should warn developers who want to protect their target applications with Invariants Monitoring: the technique may be unusable in practice.

On the other hand, False Negatives are the most pressing issue as they make this protection not secure to be used in practice to attest the integrity of target applications. False Negatives revealed in all the analysed use cases. Evidence allow the discussion to state that False Negatives are related to a lack of a precise relation between attacks that compromise the assets in the target application and the extracted likely-invariants, which is the baseline assumption for Invariants Monitoring. The trace collection and likely-invariants extraction is a statistical process that does not consider semantics, that is, the nature of assets, the security properties to satisfy on assets (i.e. integrity), and attacks paths and strategies aiming at compromising them. In other terms, there is not any clear way to assume that an

attack compromising one or more software assets will also violate likely-invariants. A clear inference among violations of invariants and attacks is an important research goal to achieve for whoever may want to develop protection techniques based on likely-invariants monitoring.

There is not any clear strategy to, at least, mitigate this issue: research should invest an enormous effort to this purpose, although there is not any hint suggesting that it may be successful. The only way that could lead to some improvements would be to support invariants with other software execution modelling tools or mechanisms in order to enhance the protection's capability of semantically describe the software behaviour. It could be worth investigating data dependency and data flow analysis techniques to improve the semantics of the inferred statements. Furthermore, Invariants Monitoring must evolve to support runtime modifications and reconfiguration (e.g. those introduced by code mobility). Combining these techniques could be a valuable direction to achieve practical application of Invariants Monitoring. Empirical assessment may be further exploited to identify human strategies that circumvent the protection, thus leading the design of an improved version of this technique.

Many of the works presented in Section 3.3 exploit likely-invariants to measure the dynamic integrity of a target. The result grasped by most of the literature's works are usually sound and seem to validate the technique. Unfortunately, in those works, likely-invariants are exploited for very particular use cases, to monitor very restricted data structures that have previously been identified by the authors. In practice, most of them based the protection on manually selected assets and invariants, thus resulting in very specialised protections that are not able to adapt to other use cases. For instance, Baliga *et al.* reports just a small set of invariants they considered to monitor very specific kernel data structures. Moreover, none of the found works discusses the Invariants Monitoring model from a higher abstraction level. The work reported in this document produced a (nearly) automatic Invariants Monitoring technique that can be applied, with minimal effort, to any target. The discussion about the Invariants Monitoring model represents the most significant part of the work reported here. The former assumption tells that a set of dynamically inferred likely-invariants can model the behaviour of an application. Most of the effort spent in investigating Invariants Monitoring was involved in validating that assumption. It has been demonstrated, with rigorous empirical work, that likely-invariants are not suited to model dynamic properties, thus to model dynamic software integrity. Hence, this work pursued a necessary in-depth study about the effectiveness of likely-invariants that was not present in literature.

In conclusion, the technique has revealed to be weak and insecure from various points of view despite what emerged in the literature. It is not able to detect a broad set of attacks. Even worse, attacks that are supposed to be detected are eventually ignored by the technique because of its inability to accurately model software behaviour.

Chapter 4

Reaction

The security analysis of static software attestation revealed a set of limitations that are intrinsic to the protection technique and the MATE scenario. Software attestation is a detection mechanism that does not include any embedded reaction mechanism. Then, even if the protection were secure and robust, there would not be any way to hinder detected attacks.

For this reason, this work proposed an extension to the basic architecture of static software attestation that improves the protection and to provide an effective reaction mechanism. The aim is to satisfy the general scheme of protection techniques: a detection system supported by a reaction mechanism. The enabling idea is to force an application to depend on a remote server in its business logic. In this way, if an attack is detected, the server can react by simply stopping to serve the client application that, consequently, would become unable to work right any more. Hence, a novel approach to anti-tampering has been proposed and published in [54]. *Reactive Attestation*, such is the name given to the proposed approach, consists of a proof-of-concept prototype that can:

1. automatically transforming a given target application to make it strictly dependent on a remote server for its normal functioning, i.e. using Client-Server Code Splitting;

2. adding tamper detection mechanism, i.e. static software attestation;

3. adding support to tamper reaction that stops the execution of tampered clients by preventing server-side execution.

The Reactive Attestation design and the consequent development have been performed in collaboration with Fondazione Bruno Kessler (FBK)[1].

[1] https://www.fbk.eu

4.1 Client-Server Code Splitting

Client-Server Code Splitting (CS-CS) transforms a program so that a portion of its functional logic executes on a remote server.

The technique was formerly designed to protect software from malicious tampering. Sensitive and critical portions of a program moved to a remote server can run securely, without being tampered with by a client-side attacker. This technique assumes that an entire function cannot always be moved to the server. A function scope might not precisely match a critical security region, that means, a code portion that requires integrity protection may be smaller than the whole code portion of the containing function. Hence, to protect code assets, it is necessary to take into consideration just a few selected parts. Moving a full function can have several side effects, e.g. updates to the program status that means synchronisation between client and server. It may require intense client-server communications, thus an excessive load of network and server. The evaluation of the portion of the client to move to the server has to take into account data and control dependencies to ensure minimal server overhead and performance impact while guaranteeing a proper level of security on the client side.

To this purpose, *barrier slicing* is the fundamental notion for CS-CS [35], which extends the concept of backward program slicing [56]. A backward slice is a subprogram that is equivalent to the original program with respect to the value of a variable at a specific statement; the pair variable-statement is called criterion. A barrier slice is computed starting from two sets of variable-statement pairs: the criterion and the barriers. All the statements in the criterion are marked. Then, control and data dependencies are backwards traversed through several iterations so that all the involved statements are marked as well. Instead, the algorithm does not traverse the dependencies on barriers. The process ends when the fixed point is reached, then the slice represents the part of the code to move away from the target application.

In the anti-tampering context, security requirements determine the criterion and the barriers. Variables in the criterion are those whose computation must be protected, and barriers are secure points where tampering is no longer an issue. If CS-CS is used to make an application server-dependent, it is sufficient to move to the server software parts that are complex enough to prevent an attacker from faking them.

When a slice is computed, a set of transformations are applied on the application. The slice is removed from the client application, and the code is replaced with synchronisation instructions. Each assignment performed to variables belonging to the criterion is replaced with a synchronisation statement that has to be reached by both client and server to allow the execution to proceed: *sync* messages exchange. The client requests values of variables to the server throughout *ask* messages, and the server requests variables' values that are on the client using *send* messages.

The whole process of transforming a program with CS-CS can be made automatic in order assist the user during the protection of his software, as done for static software attestation (see Section 3.2.3). The proposed solution aims at automatically combining and applying CS-CS and static software attestation to a target application that needs anti-tampering protection. From the research point of view, the here presented solution is intended to demonstrate that:

- software attestation can be considered as a valuable tampering detection technique for real-world programs;
- static software attestation can be used to realise robust protection methodologies, despite the underlined security issues (see Section 3.2.4), when assisted by additional reaction techniques.

4.2 Reactive Attestation

A list of requirements set the guidelines for the design of the proposed code protection approach. These requirements aim at defining robust guidelines to drive the protection system development.

Automatic protection. An automatic procedure must be able to transform the program to protect so that developers can focus their effort on developing and maintaining the program business logic instead of taking care of security and protection.

Code annotations. The developer has to explicitly indicate the code assets that need anti-tampering protection using code annotations. The protection tool must be able to identify areas to extend the protected surface in order to achieve the best security level possible; e.g. also protect the function call instead of protecting only the function implementation. This requirement is due to the lack of a method to infer the assets to protect from source code automatically.

Server dependency. The program to protect could be a stand-alone application that does not depend on the network connection, thus letting a remote tamper reaction to be easily identifiable. This kind of applications would be automatically transformed to generate a semantically equivalent application that strictly depends on a remote server to properly work, thus enabling remote reactions. The requirement sets up two main goals: turning the program into a server dependent one so that the application functionalities may be interrupted by the server and avoiding that an attacker could generate a fake server able to mimic the correct behaviour of the real server.

Accurate reactions. Tampering detection must drive the decision about reactions that have to be fast and well aimed at disturbing the target application execution. Deployed reactions should promptly block all the tampered clients while legitimate clients must not be disconnected, otherwise the protection would cause a general denial of service.

Acceptable overhead. The protection should not be too much intrusive in terms of execution time, memory and network consumption. That is, it must ensure that the protection does not impact the user experience.

The design of the Reactive Attestation protection process started from the identified set of requirements. The source code of an unprotected application is given as an input, and a set of transformations is ported to it automatically. The provided source code is annotated by the developer so that it explicitly specifies the security requirements on code portions that need integrity monitoring. The annotation syntax is detailed in Section 4.3.1.

The protection process can automatically identify the boundaries of code portions to split. The process exploits the System Dependency Graph (SDG): control and data dependencies to be affected by splitting are minimised to limit client-server interaction. The split portions are kept as small as possible to reduce overheads and server load. The split configuration is marked as annotations in the source code to protect as couples of criterion and barrier for the splitting algorithm. CS-CS is applied to generate a client-side component and a server-side one. Proper *sync*, *ask* and *send* messages are inserted into the transformed code to allow communication and synchronisation between both sides. Secondly, static software attestation transformations are applied as depicted in Section 3.2.3. The Attester is generated together with the server-side components of static software attestation, i.e. the one specified in Chapter 2.

The server-side components of the two techniques have to communicate so that software attestation can inform CS-CS whenever it detects a tamper in the target application. At runtime, the interactions between tamper detection and tamper reaction take place according to the following workflow. Static software attestation checks the integrity of the target application following the procedures described in Section 3.2. Evidence is collected at the client side by the Attester and sent back to the server side. The Verifier decrees whether the target application is still sound or not and stores the result of the verification in the central database. The history of attestation results is available to reason about reactions. The server-side reaction logic is based on an engine that processes the verification history and decides it the target has to be stopped according to a reaction policy. Policies are sets of rules that define which conditions trigger a reaction for a client application. For example, a policy may indicate that a client application has to be stopped for x minutes when the y subsequent verifications fail and to stop serving the application for $x + k \cdot t$ for all the successive failed t requests. When a reaction is triggered, the status of the target application is updated into the database.

Whenever a reaction has to take place, the system informs the CS-CS server through the database, e.g. the status of the client target application is labelled as tampered. Hence, before any client request is processed, the CS-CS server checks the application status in the database. If the application is still in the valid status, CS-CS process the request is processed and return results to the client. In case

that application is in an invalid state, thus labelled as tampered, the CS-CS server does not execute the requested code and does not reply to the request at all. Since split client and split server executions are synchronised with bidirectional data exchange, when the server-side execution stops, the client-side execution also blocks. As a consequence, the client application becomes not usable as it results in an unresponsive running program.

This scheme represents just a simple design of a robust protection mechanism that is built on top of software attestation to overcome its limitations. More complex reactions may provide more sophisticated methods. Indeed, refined reaction policies may increase the way to punish tamperings depending on its severity, the bare interruption in serving the client application can be used as last chance, and light software malfunctioning can be applied when the detected attack is considered non-critical. The presented system is designed to support the definition of sophisticated reaction policies as it involves a policy engine. For the sake of simplicity, the presented technique does not involve any sophisticated reaction as it is just meant to demonstrate the feasibility of the solution and to test the effectiveness of the obtained protection.

4.3 Implementation

The realised prototype includes CS-CS and static remote attestation modules that transform the input source code in order to deliver fully protected and working client application and server-side components. The implemented system is only able to work with source code written in C. Hence, the target application is supposed to be written in C or, at least, to have a portion that satisfies this requirement. An additional constraint, which directly derives from protection assumptions, is that the application has to run in a connected environment so that it can communicate with the server-side components over the network.

The implemented prototype is composed of four main modules: the annotation processor, the software attestation module, the Client-Server Code-Splitting module and the policy engine.

4.3.1 Annotations processing

Source code annotations are the key tool that allows the developer to specify which are the critical or sensitive assets in the application whose integrity has to be preserved. Reactive Attestation annotations extend the static software attestation ones, presented in Section 3.2.3, and are extracted and processed by the same tool.

Besides the annotation for static software attestation that allows the user to specify the Attester implementation and the memory areas to protect for each inserted Attester, Reactive Attestation introduces annotations for CS-CS. Splitting

annotations allows the user to specify criterion and barriers, and these annotations indicate the variables that the splitting algorithm has to consider. An annotation can declare a criterion by specifying `criterion` as the main parameter of the annotation. The criterion specification requires the variable name that has to be used to compute the criterion itself. The criterion annotation requires to specify the `label` parameter as well. The `label` parameter is mandatory to associate the criterion with the proper barrier. Paired with the criterion annotation comes the barrier specification annotation. The barrier annotation is specified through the `barrier` keyword. The barrier keyword parameter is the name of the variable that acts as a barrier for the splitting algorithm. Like for the criterion annotation, it is required to specify the label of the paired criterion. For instance, in the piece of code reported in Figure 4.1 has been properly annotated for Reactive Attestation. Annotation at line 1 declares a Attester for static remote attestation. The declared Attester will be in charge of monitoring the `check_license` function that is given the region annotation (line 6). Annotations at lines 22 and 26 are provided for CS-CS and respectively define the barrier involving the variable `valid_year` and the criterion that involves the `original_date` variable. The splitting algorithm computes the slices to remove from the `calculate_original` function starting from this annotated code. The protection procedure replaces the portion of the original code with synchronisation function calls. The removed code is then suited to run on the server, and whenever a reaction is needed, the server can decide to not run the code against a request from the corrupted client.

Annotations are extracted from the provided source code and then processed by the static software attestation module and CS-CS module to apply required transformations and to deliver the protected application.

4.3.2 Static Software Attestation module

The software attestation module is the same depicted in Section 3.2 and the protection is automatically applied as described in Section 3.2.3.

The extracted annotations are processed to identify the Attester to generated and to inject into the protected application. Software attestation module is independent of the CS-CS one. Indeed, software attestation transformations do not alter the source code of the target application. The CS-CS module applies most of the code modifications.

```
#annotation(declaration(RW_NORMAL, HF_SHA256,
NI_1, NG_1, MA_1, DS_1),
frequency(180),
label(a1))

#annotation(region,associated_to(a1), attest_at_startup(true))
void check_license(int day, int month, int year) {

int dd1 = calculate_original(day, month, year);
int dd2 = calculate_current()

if (dd2 - dd1 > 30)
printf("Fail\n");
else
printf("Ok\n");

}


int calculate_original(int d, int m, int y) {

#annotation(barrier(valid_year), label(s1)))
int valid_year = 0;
valid_year = check_valid();

#annotation(criterion(original_date), label(s1)))
int original_date = d + m + y + valid_year;

return original_date;
}
```

Figure 4.1: Example of source code annotated for Reactive Attestation.

4.3.3 Client-Server Code-Splitting module

The CS-CS module performs several tasks. It combines Grammatech CodeSurfer[2] and the TXL transformation framework. These tools are used to analyse the source code, to identify the portions that require to be moved onto the secure server and to apply code transformation patterns to generate the protected client application.

The implementation of Reactive Attestation is mainly a proof-of-concept. Such a prototypical approach does not face any slice optimisation concern. Indeed, if the removed code size is large, it could impact the performance of both client

[2]https://www.grammatech.com/products/codesurfer

```
1 procedure calculate-slice (criterion C, barrier B)
2 set-of-vertices := vertices-of (C)
3 set-of-barriers := vertices-of (B)
4 slice := vertices-of(C)
5 while not (is-empty(set-of-vertices))
6 predecessors := predecessors-of-vertices (set-of-vertices, DATA-CONTROL-DEPS)
7 filtered-predecessors := predecessors \ set-of-barriers
8 set-of-vertices := filtered-predecessors
9 slice := slice UNION filtered-predecessors
```

Figure 4.2: Pseudo-code of the barrier slicing algorithm.

and server application. It is due to the computation demanded to the server, which has to execute code for many clients. On the other hand, many (small or large) slices removed from the client introduces many synchronisation points with the server, that means, many interruption points in the client application due to synchronisation.

As described while discussing annotation syntax, the developer has to annotate barriers and criterion explicitly. However, literature demonstrated that algorithms are available to automatically determine barriers and criterion for a given code portion and a specified asset [12]. It is a known limitation of the prototype. The Reactive Attestation design deliberately ignored it to privilege the security aspects rather than the engineering and implementation tasks.

Once barriers and criteria are available, the computation of the slice and all the program transformation steps are fully automated. The input of the tool are the annotations extracted from source code, while the output consists of the sliced client and the corresponding server-side slice component.

The component that is in charge of computing the barrier slice is implemented as a CodeSurfer analysis script. The implemented algorithm precisely calculates the portion of code that represents the barrier slice according to the annotation configuration and the code to protect. The give source code is analysed by CodeSurfer to calculate the system dependence graph (SDG). The slicing algorithm iterates by querying SDG to extract the barrier slice.

Figure 4.2 shows the pseudo-code of barrier slicing. Input parameters are the slicing criterion C and the barrier B. These parameters are converted into vertices of the SDG and stored in two local variables, *set-of-vertices* and *set-of-barriers* respectively (line 2, 3). Then, the barrier slice, stored in variable *slice*, is initialised to the vertices that represent the criterion C (line 4). Then, the algorithm iterates by adding vertices to the slice set and stops when it is not able to add any new vertex to the slice, that is the fixed point is reached. More specifically, in the first iteration, the algorithm queries all the predecessors of the current set of vertices, i.e. the criterion C, by following data and control dependencies backwards in the SDG. Possible barriers are filtered out from the set of predecessors. The propagation of

dependencies stops when a barrier is found; then, the procedure does not include vertices at that barrier in the slice. After the filtering, the resulting set of vertices becomes the set of vertices for the next iteration. The algorithm stops and returns the final barrier slice when it does not find any new predecessor.

Once the procedure obtains the slice, it applies a set of program transformation rules. TXL rewriting transformations are used to the parse the abstract syntax tree of the program and to modify the source code in order to create a new compilation unit for the server side component and to transform the client program so that it only works when coupled with the corresponding server.

The implementation activities involved the author for what concerned the development of static software attestation module while FBK contributors developed the CS-CS module. The final integration of the two modules has been carried out by the author in collaboration with the FBK staff[3].

4.3.4 Policy engine

The central database has the same structure described in Section 2.7. Additional data for CS-CS management and communications between the two modules extends the static software attestation database structure.

A `reaction_status` table reports the allowed values for the overall application status, depending on the reaction policy. This information is brought by an additional table, namely the Reaction Statuses table.

A `reaction` table stores the overall status of clients as established by the reaction policy engine. In practice, it associates a Reaction Status value to a target application ID.

A `policy` table stores the association between a protected application and the reaction policy that must be enforced on it. Enforceable policies are predefined and used as constant values.

The reaction engine has been trivially implemented just to prove the feasibility of the enabling idea. Such an engine is implemented as a set of processes, one for each used policy. Each process accesses the database and interprets the last attestation results to decide and whether it is needed or not and hot to react according to the implemented policy. The prototype implements three sample policies: "stop serving a tampered application", "stop serving a tampered application until its reboot", and the "stop serving for x minutes an application whose last y attestations failed and to stop serving the application for $x + k \cdot t$ for all the successive failed t verifications".

[3] An acknowledgement is due to thank **Andrea Avancini** for the effort invested in making static software attestation and CS-CS mechanisms work together.

4.4 Experimental validation

A set of experimental assessments have been performed to validate the procedure and to verify the prototype effectiveness. The empirical evaluation has been set to address the following research questions.

1. **RQ1 – Accuracy**: is Reactive Attestation (i.e. static software attestation) effective in detect tampering?

2. **RQ2 – Overhead**: how large is the overhead given by Reactive Attestation?

The first research question aims at measuring the accuracy of the prototypical protection technique in detecting real cases of code modifications. In other words, the assessment aimed at verifying that Reactive Attestation can block tampered applications while not impacting legitimate ones. The second research question is intended to investigate the cost of the protection technique, in terms of execution delay for client applications compared to the execution of the original application.

A set of metrics has been set to measure the effectiveness of the developed system, thus to address the RQ1 research question.

True Positives (TP), it counts the number of tampered targets that get *correctly* blocked by the protection;

False Positives (FP), it counts the number of genuine targets that get *incorrectly* blocked by the protection;

True Negatives (TN), it counts the number of genuine targets that are *correctly* let to execute by the protection;

False Negatives (FN), it counts the number of tampered targets that get *incorrectly* let to execute by the protection.

The defined metrics are key metrics in information retrieval to measure the performance of a classifier to classify documents correctly, so they are useful to measure the ability of the Reactive Attestation prototype to tell a legitimate target from an unauthorised one. The assessment aims at maximising True Positives and True Negatives in order to reach the best effectiveness of the correct functioning of the protection. On the other hand, minimising the number of False positives and False Negatives would improve the protection as a reduction of its failures in tamper detection.

For what concerns overhead assessment, the metrics considered for evaluation are the following ones.

Memory (MEM), the amount of memory required to execute the original and the protected program. Memory consumption is measured using the `time` utility[4]. The time command runs a program, and displays information about the resources, like memory and time, consumed by that program.

[4]https://linux.die.net/man/1/time

Execution time (TIME) is the time spent to run the program. As done with MEM, TIME is measured using the `time` system utility.

Network usage (NET) is the amount of data exchanged by the program in a complete run. NET is directly measured at server-side by the static software attestation and CS-CS components.

One can expect to achieve an attack surface reduction when Reactive Attestation is applied so that an attacker could tackle fewer points in the program. As a drawback, the introduced transformations impact the overall application performance, in particular on the execution time and memory occupation. Hence, the defined metrics aim at estimating the magnitude of this impact.

The empirical assessment has been conducted on a use case application, that is a license check application, namely *License*. License is an Android Java application whose critical parts resides in native C library. These critical parts implement the routine in charge of checking the validity of the provided license number and consequently enable or not further software components. Reactive Attestation is deployed to the native part of the application as it is the real critical asset that needs protection. The protected code is composed of two functions for a total of 105 LOC. One function verifies the license code, and it is monitored by software attestation while CS-CS protect the decoding function used by the previous one. If the decode function is inhibited, CS-CS acts as a server-side reaction that prevents the application to work correctly. It this way, the two modules of Reactive Attestation fully protect the entire security critical code.

The assessment involved seven different tampered protected versions of the application and one protected untampered version. The tampering has been ported by rewriting the binary code of the built target. For instance, tampering included: skipping the check function call as it is overwritten with a NOP instruction, forcing a date as a current date in the license period, and altering the license expiration date. The assessment procedure slight modified the application so that the original single license check became a loop of 100 consecutive checks. It was the only way to observe the application run and grasp valuable data. Indeed, this workaround provided measurable executions that were not possible with the original application because its execution was too fast. The evaluation executed each application version 100 times.

RQ1 – Accuracy. Results about True/False Positives and Negative are reported in Table 4.1. Reactive Attestation correctly reports the untampered version as a True Negative, such as it is executed, checked and allowed to execute as malicious modifications are not detected. All seven tampered versions were correctly classified as True Positives. Hence, according to the experiment setting, Reactive Attestation is totally able to identify and grant execution to legitimate clients in 100% of the cases. It was also able to identify and stop, with 100% of success, all the tampered clients.

Consideration: Reactive Attestation is effective as it blocks all the execution

of malicious targets. The prototype did not interfere with the execution of legitimate and untampered clients. It is worth to notice that the accuracy observation is valid on the considered use case, which is small and (maybe) not representative of real-world applications. In addition, Reactive Attestation is intrinsically subject to all the vulnerabilities that affect static software attestation (as reported in Section 3.2.4).

Variant	TP	FP	TN	FN
Protected	-	-	✓	-
Tampered 1	✓	-	-	-
Tampered 2	✓	-	-	-
Tampered 3	✓	-	-	-
Tampered 4	✓	-	-	-
Tampered 5	✓	-	-	-
Tampered 6	✓	-	-	-
Tampered 7	✓	-	-	-
Overall	7	0	1	0

Table 4.1: Effectiveness of Reactive Attestation.

RQ2 – Overhead. Table 4.2 reports the results about measured overhead. For each considered usage scenarios (column 1), the Table reports mean and standard deviation of absolute values of Memory (MEM, column 2), execution time (TIME, column 3) and network usage (NET, column 4). The scenario considered to assess the overhead research question is represented by (1) the vanilla application run with valid and invalid input license code, and (2) the protected (untampered) application run with valid and invalid license input code.

Memory usage increases about six times when protection is applied. It is due to the setup of the network communication infrastructure used by Reactive Attestation, to the memory used by CS-CS communications and to the size of the inserted Attester and its ADS. All these components, except for the ADS, introduce a constant memory overhead that does not depend on the application or protection configuration. The size of ADS linearly depends on the number of monitored code regions. Execution time is the most problematic impacted metric. Indeed, the protection slows down the average execution by two orders of magnitude. The network load revealed to be constant. The data reported by column 4 of Table 4.1 is the exact amount of network data that Reactive Attestation exchanges for a single application run, without considering the network consumed to establish the connection between the two endpoints.

Consideration 1: Reactive Attestation requires time as a considerable execution time increase revealed from the vanilla application to the protected one. Although for the assessed use case the execution time overhead is quite not perceivable from

a user experience point of view, it is not possible to ignore the issue for the general case. It is worthy of considering that for the studied use case, the full code has been protected with Reactive Attestation instead of focusing on specific variables or assets. Then, it could be possible to mitigate the alarming impact on execution overhead by wisely determine the size of the protected code. Focusing only on small portions of the application could drastically reduce the execution impact. However, in the use case, 105 lines of code were protected, which is roughly comparable to the dimension of a security-critical area in real-world applications. In the case of bigger applications, the execution time impact may be most probably dominated by the regular execution time, which is usually much larger than a few microseconds.

Consideration 2: Reactive Attestation has a small network impact. Collected evidence tells that the data exchanged over the network by the protection prototype are very limited so that it can be used to protect application in mobile contexts. It has to be taken into account that the network load strongly depends on the functionalities and the nature of the protected code. Applying CS-CS on a functionality that is computationally intensive and involves lots of variables may highly increase the network usage. Hence, the protection configuration has to be carefully chosen in order to avoid severe network impact.

In addition, performance overhead depends on how often the target execution encounters the split points and on the complexity of the split code. Nevertheless, to avoid performance issues related to CS-CS, the parts to split should be selected by analysing the execution traces in order to quantify the most executed parts of the target application. Considering also this kind of information in the slicing process may help to maintain the overhead to an acceptable level. Finally, performance represents an open issue and an important point that has to be addressed to bring this solution to a practically usable level, but this is out of the scope of the work presented here. Indeed, it aimed just at demonstrating that software attestation can be combined with other protection techniques to fit its security lacks, complete its detection ability with reaction facilities and obtain robust protection.

In conclusion, Reactive Attestation is a quite successful attempt to integrate a reaction mechanism such as CS-CS and a tamper detection method such as static software attestation. It can automatically protect an application starting from its source code. The only effort requested to the developer is to annotate the code to specify the desired security requirements. The application is then transformed to be made server-dependent, thus to have a way to punish tampered applications. Experimental validation of the prototypical approach demonstrated its effectiveness accuracy but underlined performance issues that have to be faced from an engineering point of view to make the methodology practically usable.

This work demonstrates that although software attestation comes with a set of limitations and security issues, it can be combined with other protections to achieve adequate protection with the right security level. The whole discussion presented in this work suggests that software attestation techniques are not very robust and

Variant (scenario)	MEM		TIME		NET	
	Mean (kB)	SD	Mean (ms)	SD	Mean (B)	SD
Original (valid license)	2,192	0	0.17	0.006	-	-
Original (invalid license)	2,176	0	0.18	0.004	-	-
Protected (valid license)	13,715	145.540	84.41	0.290	392	0
Protected (invalid license)	13,960	33.598	85.06	0.323	392	0

Table 4.2: Overhead of Reactive Attestation.

not very effective to protect real-world applications. However, software attestation has been demonstrated to be usable as a starting point to build more sophisticated security mechanisms for software protection that harden programs and effectively hinder attacker tasks. For instance, the cloning attack presented in Section 3.2.4 represents a severe security threat for static software attestation itself. However, cloning attacks can be mitigated by coupling static software attestation with other software protection techniques as happens with CS-CS in Reactive Attestation. Indeed, if a protection technique like Reactive Attestation removes sensitive and strategic pieces of code from the target, they cannot be tampered with on client-side. Then the power of cloning attacks is effectively reduced (as for other client-side-based attacks).

Chapter 5

Conclusions

The work reported in this document aimed at analysing Software Attestation as a complete method for software security. The general model was designed as an abstract architecture that suits robust protection mechanisms for any kind software attestation. It defines attestation procedures and protocols with a high abstraction level in order to describe the procedures and leave internal customisations to the actual implementation of the technique. Furthermore, the attestation model is robust as it is resilient, by design, to the well known attacks, e.g. MitM, sniffing or replay attacks, and unpredictable as it involves random processes and challenge-based mechanisms. Moreover, the presented architecture is modular and allows any security-skilled developer to implement new functionalities or to extend the designed features to adapt the protection system to his needs.

Afterwards, investigation on the detection ability of software attestation was performed. General requirements for a detection system ware settled in order to drive any instantiation of the software attestation model. The analysis proceeded by considering two essential software features: static and dynamic properties. Static software attestation aims at monitoring the structural integrity of a program. The nature of the assets whose integrity can be checked was defined as well as the procedures to characterise the abstract software attestation model to retrieve integrity evidences. Consequently, a practical implementation of static software attestation was reported to show what are the main choices to make and the issue to face when this kind of protection is implemented and deployed in practice.

Dynamic software attestation was presented as a second instantiation of the abstract model. It aims at monitoring the integrity of software behavioural features, that are features that describe the software execution instead of its structure. Reference requirements were defined and, according to them, it was presented the way to customise the software attestation model. Invariants Monitoring was presented as a dynamic software attestation solution that model application behaviour by means of automatically inferred likely-invariants. The analysis studied the modelling properties of likely-invariants and assessed their effectiveness for protection

purposes. An Invariants Monitoring working prototype was implemented both to provide a practical evidence of the applicability of dynamic software attestation and to experimentally asses its protection effectiveness.

The presented flavours of software attestation were designed to be automatically applicable on arbitrary software applications. Hence, two distinct toolchains and workflows were presented to emphasize that characterisation is tightly related to the internal details of each attestation instantiation but the abstract model is valid regardless of the low level details and can be automatically applied.

To complete the security discussion about static and dynamic software attestation, security analyses were performed on both the presented mechanisms. Static software attestation is subject to a set of issues that are related to the architectural choices made during its instantiation and intrinsic of its the model. The most critical issues are represented by cloning attacks and non-structural runtime modifications, which seem to be impossible to overcome just by improving the technique. Most of the other issues can be resolved by enough engineering effort or by combining static software attestation with other techniques. To this purpose, Reactive Attestation was presented to show how static software attestation can be used to drive reactions provided by tamper reaction techniques. The results grasped in reactive attestation study can be generalised to assert that software protections can be used in combination to achieve higher software security levels.

As opposite to the positive outcomes observed during the static software attestation analysis, the analysis performed on Invariants Monitoring led to assert that the technique should not be used for software security purposes. Likely-invariants were a promising tool to model the software behaviour, then a broad investigation on likely-invariants have exposed critical limitations. Indeed, invariants inferring procedures collect non-precise invariants or, at least, they collect invariants that are not valuable to describe software execution aspects. Use cases were involved in the analysis to better analyse the effectiveness of Invariants Monitoring and to try to overcome the encountered limitations, but also in this case the technique revealed to be fallacious. The most harmful and dangerous issues are related to false negatives that generate from the complete absence of any inference between corrupted application and violated invariants. Consequently, the protection is unable to detect attacks, thus identifying as genuine a tampered application.

Concluding, software attestation is a valuable solution for tamper detection as its model can be applied to all the modern distributed scenarios. Software attestation can be customised to fit the most disparate needs. The presented design is completely independent from any underlying hardware or software architecture and can be applied to any kind of software application. Moreover, it is compatible with other protection techniques and represents a valid building block for robust and complex protection methodologies.

Limitations are introduced by particular instantiations of the abstract model. In particular, the definition model used to describe the assets is a crucial point when

software attestation specialisation is designed. Indeed, conceptual errors introduced by the asset model may lead to severe drawbacks if the model was not well investigated, as it happened for Invariants Monitoring. Static software attestation has revealed to be successful even if it needs support by other techniques while dynamic software attestation has emerged to be more critical. Dynamic software attestation involves more complex software features, thus more complex modelling methods to describe software states. Hence, future directions in software attestation research should mainly be focused on better structures to model software aspects and features. The most challenging limitation is about modelling the software behaviour and describing its execution using tools that are semantically expressive and robust from the security point of view.

Bibliography

[1] Martı́én Abadi et al. "Control-flow integrity principles, implementations, and applications". In: *ACM Trans. Inf. Syst. Secur.* 13.1 (2009), pp. 1–40. ISSN: 1094-9224. DOI: http://doi.acm.org/10.1145/1609956.1609960.

[2] Tigist Abera et al. "C-FLAT: control-flow attestation for embedded systems software". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 743–754.

[3] Adnan Akhunzada et al. "Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions". In: *Journal of Network and Computer Applications* 48 (2015), pp. 44–57.

[4] Masoom Alam et al. "Model-based Behavioral Attestation". In: *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*. SACMAT '08. Estes Park, CO, USA: ACM, 2008, pp. 175–184. ISBN: 978-1-60558-129-3. DOI: 10.1145/1377836.1377864. URL: http://doi.acm.org/10.1145/1377836.1377864.

[5] Frederik Armknecht et al. "A security framework for the analysis and design of software attestation". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 1–12.

[6] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. "Automatic inference and enforcement of kernel data structure invariants". In: *Computer Security Applications Conf. 2008. ACSAC 2008. Annual*. IEEE. 2008, pp. 77–86.

[7] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. "Detecting kernel-level rootkits using data structure invariants". In: *IEEE Transactions on Dependable and Secure Computing* 8.5 (2011), pp. 670–684.

[8] Cataldo Basile, Stefano Di Carlo, and Alberto Scionti. "FPGA-based remote-code integrity verification of programs in distributed embedded systems". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.2 (2012), pp. 187–200.

[9] Dirk Beyer et al. "Path invariants". In: *ACM Sigplan Notices*. Vol. 42. 6. ACM. 2007, pp. 300–309.

[10] Bruno Blanchet et al. "A static analyzer for large safety-critical software". In: *ACM SIGPLAN Notices*. Vol. 38. 5. ACM. 2003, pp. 196–207.

[11] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. "From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing". In: *Proc. of the 2006 int. symposium on Software testing and analysis*. ACM. 2006, pp. 169–180.

[12] Mariano Ceccato et al. "Trading-off security and performance in barrier slicing for remote software entrusting". In: *Automated Software Engineering* 16.2 (2009), pp. 235–261.

[13] Liqun Chen et al. "A Protocol for Property-based Attestation". In: *Proceedings of the First ACM Workshop on Scalable Trusted Computing*. STC '06. Alexandria, Virginia, USA: ACM, 2006, pp. 7–16. ISBN: 1-59593-548-7. DOI: 10.1145/1179474.1179479. URL: http://doi.acm.org/10.1145/1179474.1179479.

[14] Ernie Cohen et al. "VCC: A practical system for verifying concurrent C". In: *Int. Conf. on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 23–42.

[15] George Coker et al. "Principles of remote attestation". In: *International Journal of Information Security* 10.2 (2011), pp. 63–81.

[16] Victor Costan and Srinivas Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016.086 (2016), pp. 1–118.

[17] Flaviu Cristian. "Exception handling and software fault tolerance". In: *IEEE Transactions on Computers* 31.6 (1982), pp. 531–540.

[18] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. "DySy". In: *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th Int. Conf. on*. IEEE. 2008, pp. 281–290.

[19] Nelly Delgado, Ann Q Gates, and Steve Roach. "A taxonomy and catalog of runtime software-fault monitoring tools". In: *IEEE Transactions on software Engineering* 30.12 (2004), pp. 859–872.

[20] Michael D Ernst et al. "Dynamically discovering likely program invariants to support program evolution". In: *IEEE Transactions on Software Engineering* 27.2 (2001), pp. 99–123.

[21] Michael D Ernst et al. "The Daikon system for dynamic detection of likely invariants". In: *Science of Computer Programming* 69.1 (2007), pp. 35–45.

[22] Tal Garfinkel et al. "Terra: A virtual machine-based platform for trusted computing". In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003, pp. 193–206.

[23] David Gries. *The Science of Programming*. New York: Springer-Verlag, 1981. ISBN: 978-1-4612-5983-1. DOI: 10.1007/978-1-4612-5983-1.

[24] Gisle Grimen, Christian Mönch, and Roger Midtstraum. "Tamper Protection of Online Clients through Random Checksum Algorithms". In: *Information Systems Technology and its Applications, 5th International Conference ISTA'2006, May 30-31, 2006, Klagenfurt, Austria*. 2006, pp. 67–79. URL: http://subs.emis.de/LNI/Proceedings/Proceedings84/article4304.html.

[25] Trusted Computing Group. *TPM Specification version 1.2. Parts 1–3*. https://trustedcomputinggroup.org/resource/tpm-main-specification. [Online; accessed 19-July-2018]. 2003.

[26] Sudheendra Hangal and Monica S Lam. "Tracking down software bugs using automatic anomaly detection". In: *Proc. of the 24th int. conf. on Software engineering*. ACM. 2002, pp. 291–301.

[27] Sudheendra Hangal et al. "IODINE: a tool to automatically infer dynamic invariants for hardware designs". In: *Proc. of the 42nd annual Design Automation Conf.*. ACM. 2005, pp. 775–778.

[28] Charles Antony Richard Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[29] Trent Jaeger, Reiner Sailer, and Umesh Shankar. "PRIMA: policy-reduced integrity measurement architecture". In: *Proceedings of the eleventh ACM symposium on Access control models and technologies*. ACM. 2006, pp. 19–28.

[30] Mariusz H Jakubowski, Chit Wei Nick Saw, and Ramarathnam Venkatesan. "Tamper-tolerant software: Modeling and implementation". In: *International Workshop on Security*. Springer. 2009, pp. 125–139.

[31] J-M Jazequel and Bertrand Meyer. "Design by contract: The lessons of Ariane". In: *Computer* 30.1 (1997), pp. 129–130.

[32] Chongkyung Kil et al. "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence". In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE. 2009, pp. 115–124.

[33] Gerwin Klein et al. "seL4: Formal verification of an OS kernel". In: *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 207–220.

[34] René Korthaus et al. "A Practical Property-based Bootstrap Architecture". In: *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*. STC '09. Chicago, Illinois, USA: ACM, 2009, pp. 29–38. ISBN: 978-1-60558-788-2. DOI: 10.1145/1655108.1655114. URL: http://doi.acm.org/10.1145/1655108.1655114.

[35] Jens Krinke. "Barrier Slicing and Chopping". In: *SCAM*. 2003, pp. 81–87.

[36] Xiao-Yong Li, Chang-Xiang Shen, and Xiao-Dong Zuo. "An Efficient Attestation for Trustworthiness of Computing Platform". In: *Proceedings of the 2006 International Conference on Intelligent Information Hiding and Multimedia*. IIH-MSP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 625–630. ISBN: 0-7695-2745-0. DOI: 10.1109/IIH-MSP.2006.48. URL: http://dx.doi.org/10.1109/IIH-MSP.2006.48.

[37] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezze. "Towards self-protecting enterprise applications". In: *Software Reliability, 2007. ISSRE'07. The 18th IEEE Int. Symposium on*. IEEE. 2007, pp. 39–48.

[38] Job Noorman et al. "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base". In: *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 2013, pp. 479–498.

[39] Kazuomi Oishi and Tsutomu Matsumoto. "Self destructive tamper response for software protection". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM. 2011, pp. 490–496.

[40] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. "vTPM: virtualizing the trusted platform module". In: *Proc. 15th Conf. on USENIX Security Symposium*. 2006, pp. 305–320.

[41] Jeff H Perkins et al. "Automatically patching errors in deployed software". In: *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 87–102.

[42] Nick L Petroni Jr et al. "Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor." In: *USENIX Security Symposium*. San Diego, USA. 2004, pp. 179–194.

[43] Ahmad-Reza Sadeghi and Christian Stüble. "Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms". In: *Proceedings of the 2004 Workshop on New Security Paradigms*. NSPW '04. Nova Scotia, Canada: ACM, 2004, pp. 67–77. ISBN: 1-59593-076-0. DOI: 10.1145/1065907.1066038. URL: http://doi.acm.org/10.1145/1065907.1066038.

[44] Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. "Property-Based TPM Virtualization". In: *Information Security: 11th International Conference, ISC 2008, Taipei, Taiwan, September 15-18, 2008. Proceedings*. Ed. by Tzong-Chen Wu et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–16. ISBN: 978-3-540-85886-7. DOI: 10.1007/978-3-540-85886-7_1. URL: http://dx.doi.org/10.1007/978-3-540-85886-7_1.

[45] Reiner Sailer et al. "Design and Implementation of a TCG-based Integrity Measurement Architecture." In: *USENIX Security Symposium*. Vol. 13. 2004, pp. 223–238.

[46] Arvind Seshadri et al. "Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems". In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: ACM, 2005, pp. 1–16. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1095812. URL: http://doi.acm.org/10.1145/1095810.1095812.

[47] Arvind Seshadri et al. "SWATT: Software-based attestation for embedded devices". In: *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*. IEEE. 2004, pp. 272–282.

[48] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. "Bind: A fine-grained attestation service for secure distributed systems". In: *null*. IEEE. 2005, pp. 154–168.

[49] Diomidis Spinellis. "Reflection As a Mechanism for Software Integrity Verification". In: *ACM Trans. Inf. Syst. Secur.* 3.1 (Feb. 2000), pp. 51–62. ISSN: 1094-9224. DOI: 10.1145/353323.353383. URL: http://doi.acm.org/10.1145/353323.353383.

[50] Gang Tan, Yuqun Chen, and Mariusz H Jakubowski. "Delayed and controlled failures in tamper-resistant software". In: *International Workshop on Information Hiding*. Springer. 2006, pp. 216–231.

[51] Paul C Van Oorschot, Anil Somayaji, and Glenn Wurster. "Hardware-assisted circumvention of self-hashing software tamper resistance". In: *IEEE Transactions on Dependable and Secure Computing* 2.2 (2005), pp. 82–92.

[52] Alessio Viticchié, Cataldo Basile, and Antonio Lioy. "Remotely assessing integrity of software applications by monitoring invariants: Present limitations and future directions". In: *International Conference on Risks and Security of Internet and Systems*. Springer. 2017, pp. 66–82.

[53] Alessio Viticchié et al. "On the impossibility of effectively using likely-invariants for software attestation purposes." In: *JoWUA* 9.2 (2018), pp. 1–25.

[54] Alessio Viticchié et al. "Reactive attestation: Automatic detection and reaction to software tampering attacks". In: *Proceedings of the 2016 ACM Workshop on Software PROtection*. ACM. 2016, pp. 73–84.

[55] Jinpeng Wei et al. "Modeling the runtime integrity of cloud servers: a scoped invariant perspective". In: *Privacy and Security for Cloud Computing*. Springer, 2013, pp. 211–232.

[56] Mark Weiser. “Program Slicing”. In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE ’81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. ISBN: 0-89791-146-6. URL: http://dl.acm.org/citation.cfm?id=800078.802557.