

On the impossibility of effectively using likely-invariants for software attestation purposes

Alessio Viticchié^{1*}, Cataldo Basile¹, Fulvio Valenza^{1,2}, and Antonio Lioy¹

¹*Politecnico di Torino, Dip. Automatica e Informatica, c. Duca degli Abruzzi 24, Turin, Italy*

²*CNR-IEIIT, c. Duca degli Abruzzi 24, Turin, Italy*

first.last@polito.it

Abstract

Invariants monitoring is a software attestation technique that aims at proving the integrity of a running application by checking likely-invariants, which are statistically significant predicates inferred on variables' values. Being very promising, according to the software protection literature, we developed a technique to remotely monitor invariants. This paper presents the analysis we performed to assess the effectiveness of our technique and the effectiveness of likely-invariants for software attestation purposes. Moreover, it illustrates the identified limitations and our attempts to improve the detection abilities of this technique. Our results suggest that, although further studies and future results might increase its effectiveness and reduce the side effects, software attestation based on likely-invariants is not yet ready for the real world. Software developers should be warned of these limitations, if they would be tempted by adopting this technique, and companies developing software protections should not invest in development without investing in further research too.

Keywords: invariants monitoring, software attestation, likely-invariants, software protection

1 Introduction

Software attestation is methodology used to verify that a program running on another system is behaving as expected. Software attestation techniques check the program memory against modifications made by malicious code and decree about the program integrity [1]. As opposed to the remote attestation procedure defined by the Trusted Computing Group [2, 3], software attestation techniques do not rely on secure hardware (i.e., a Trusted Platform Module) to establish a root of trust and compute the integrity proofs [4]. They rather build on features of the software to monitor, such as execution time [1] or software of environment properties [5]. Therefore, software attestation is better suited for scenarios where hardware security features are not available, like often happens for portable devices, embedded systems, and Internet of Things devices.

Invariants monitoring (IM) has been proposed in literature as a valid approach to software attestation [6]. Invariants, borrowed from software engineering, are logical assertions that are held to be true during the execution of (some parts of) the application. Introduced and used for maintenance and verification purposes (assertions, programming by contract [7, 8, 9]), invariants have been also used to detect programming errors and incorrect implementations [10, 11].

Unfortunately, user-specified invariants are often nearly absent in applications [12], hence software developers wanting to maintain their software and defenders wanting to protect selected code areas from tampering, may experience an insufficient number of invariants. For this reason, researchers have proposed the use of dynamically-extracted likely-invariants, simply named in this paper as *likely-invariants*.

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), 9:2 (June 2018), pp. 1-25

*Corresponding author: Politecnico di Torino, Dip. di Automatica e Informatica, Corso Duca degli Abruzzi 24, 10129 Torino, Italy, Tel: +39-0110907192

All the previous works in literature about IM resorted to likely-invariants. Likely-invariants, as opposed to the “true” invariants described before, are inferred from the analysis of traces collected by executing the applications on selected inputs [12]. Usually, several likely-invariants can be inferred for every part of the application whose integrity needs to be ensured.

Being very promising on paper, we have designed and implemented an IM technique within the context of the EC funded project ASPIRE¹. In our design, a client-side Attester is embedded in the protected application and sends to a trusted remote server the values of selected variables, among the ones in memory. On the server, an integrity Verifier uses the received values to check likely-invariants and establish trustworthiness of the target application. Moreover, we have created a tool chain that identifies the likely-invariants in a program and generates both the client-side protected program and the server-side verification logic. All the instrumentation phases are automatically performed by the tool chain, apart from the selection of the likely-invariants to monitor and the adaptation of the compilation options, which are human assisted due to the variety of compiling tools available (and because developers and protectors involved in ASPIRE wanted to control them explicitly).

Unfortunately, when using likely-invariants to protect the project use cases (both benchmark open source applications and programs from the industrial partners), we have soon experimented several limitations. Therefore, we investigated if the identified limitations are structural of IM (i.e., if likely-invariants are not suitable to protect target applications), or simply related to our implementation (i.e., engineering effort may render IM a valid software attestation technique).

Hence, the major contributions of this paper are the analysis of the impact and the evaluation of the effectiveness of IM, which includes the identification and categorisation of the main limitations, and the analysis and assessment of the detection abilities of likely-invariants.

For what concerns the limitations, we have first identified *technological limitations*, mainly depending on the tool that extracts the invariants and in some cases related to our implementation. For instance, the most known and used tool, Daikon [13], does not infer invariants on matrices, structures, and inner scope variables. These limitations limit the detection ability of this technique but can be overcome with proper engineering effort.

We have also noted limitations *intrinsic limitations*, which are related to the use of likely-invariants and do not depend on any technological or implementation aspect. For instance, likely-invariants are true for the analysed traces but they are not true in general for all the possible application executions. Therefore, for protection purposes, likely-invariants may generate *false positives*, i.e., violations of invariants that are not related to attacks. The presence of false positives, like for Intrusion Detection System, requires human effort to analyse individual cases and may severely affect the management effort associated to this protection.

Furthermore, additional limitations are related to the use of likely-invariants for protection purposes. The assumption that a defender could tell compromised applications from original ones by collecting runtime values of variables from the memory and use them to verify invariants has never been formally proven. Indeed, the research illustrated in this paper shows what we have experienced: this assumption is too often false, thus not acceptable in practice. Indeed, both in the test cases we have used during the protection development and the industrial use cases of ASPIRE, we have mounted attacks that were not discovered by the IM technique. We have defined these cases as *false negatives*, i.e., attacks that compromise applications without violating invariants. Moreover, from an empirical assessment we have conducted with students, we have proven that also amateur hackers can mount attacks that alter code and memory values and are not detected with IM. We deduced that the foundational assumption of invariants monitoring is not true in general. Thus the effectiveness of invariants in detecting each attack against the application assets needs to be empirically proven. Even worse, every single version of the target

¹<https://aspire-fp7.eu/>

application would require a similar empirical assessment.

Therefore, given the practical downsides in management and security of this technique, practitioners are not suggested to use it to protect real world critical software applications. Moreover, software companies interested in developing protections techniques, which are recently more attracted by integrity checking methods, are strongly discouraged in approaching IM without planning extensive research or until some major assurance and correlations to software behavior are not set, which is in our opinion quite unlikely.

An initial version of the paper has been already published [14]. It listed and categorized all the limitations found against IM. This paper extends the initial version and presents how we tried to overcome some of the identified limitations and the corresponding improvements to our IM technique. However, we have not addressed all of the limitations, especially the ones that could be solved with engineering effort by developers interested in investing in this protection. Indeed, our goal was determining if IM can be used for protection purposes, thus, this paper focuses on more foundational limitations of IM and concentrates on the two most important limitations we have found: false positives and false negatives. These limitations have been assessed on a set of use cases. Then, we have tried to generalize on the general effectiveness of IM. The results presented in this paper are also very different from the ones in the previous version. While we claiming for more research to prove the effectiveness of this techniques, this paper shows a little less hope to save this technique. Either major advancement on attestation are performed (e.g., on dynamic aspects of the program execution) together with formal models of invariants, or IM will never be usable in practice.

The paper is organized as follows. Section 2 introduces the background on invariants, likely-invariants, and how invariants have been used for software protection purposes. Section 3 presents the IM technique we have implemented. Section 4 presents Daikon and the technological limitations it introduces.

Section 5 introduces the use cases and the likely-invariants extracted while Section 6 evaluates their effectiveness for software protection purposes according to extraction strategies. Section 7 discusses the major limitations limitations of invariants monitoring. Section 8 and 9 shows our tool chain that instruments a target application to be protected with IM. Finally, Section 10 draws conclusions and sketches future works.

2 Background on Invariants and Related Works

Invariants are known in computer science since long time, as they were used to identify program bugs before they revealed during application working, and to assist development and maintenance [7]. As Gries demonstrated, a program can be formally derived from its specifications [8], hence software engineers defined axioms that describe correct executions and properties in terms of constraints. Several works in literature exploited invariants as axioms to identify and locate faults and problems in software implementation [15, 16, 17, 18, 19]. Indeed, in these works invariants were explicitly deduced from the theoretical specifications of the program, then checked against the actual implementation.

In practice, it could happen that invariants are absent or too little to be useful. Therefore, researchers proposed to dynamically extract invariants from target applications by observing the execution of an application and inferring a set of relations, which are then used as invariants. Dynamically extracted invariants are not axioms, rather constraints that can fail, by chance, even if the program is sound. Hence, the term *likely invariants* has been introduced to identify these empirically deduced constraints.

One of the most widespread tools for likely invariants detection is Daikon, which is free and open source². Daikon tests potential invariants against observed runtime values, taken from traces collected

²Daikon is the subject of a number of publications, which is maintained at <https://plse.cs.washington.edu/daikon/pubs/>

through a front-end instrumentation of the target application [12, 13]. Daikon uses a statistical approach to minimize invariants that may generate false positives. That is, only the true relations whose probability to be a mere coincidence is under a user-definable threshold become invariants. Moreover, Daikon can improve its effectiveness with abstract type interpretation [20].

Daikon has been used by several authors for software maintenance purposes. Xie *et al.* used Daikon to extract invariants for automatic unit test generation without any *a priori* specification [21]. Csallner *et al.* proposed DSD-Crasher, a tool that automatically finds software bugs by combining dynamic analysis (i.e., likely invariants inferred with Daikon) with static analysis and dynamic verification [22]. Schuler *et al.* proposed a tool to infer class contracts with Daikon [23]. Schiller *et al.* proposed VeriWeb, a web-based IDE that uses Daikon to infer clauses used to help users when they have to write verifiable specifications [24]. Sahoo *et al.* exploit Daikon-generated likely invariants to locate elements in the code that may lead to failures. Likelihood failure is assessed via dynamic backward slicing and heuristic filtering [25]. Lemieux *et al.* introduced Texada, a tool for extracting specifications in linear temporal logic that once combined with Daikon infers likely data-temporal properties [26].

Beside Daikon, several other tools and methodologies have been developed for likely invariant detection. DIDUCE by Hangal *et al.*, performs dynamic invariants detection on online Java applications at run time [11]. It automatically detects program errors and their causes (e.g., errors in handling corner cases, errors in inputs, and misuse of APIs) by analysing the detected invariants. Csallner *et al.* proposed DySy, a dynamic invariants detector that combines concrete executions and symbolic executions of test cases to deduce *symbolic invariants*, which not only depend on execution traces but also on text structure of the program [27]. Along with academic progresses, industrial invariants detection solutions have appeared: Ackermann *et al.* proposed a tool that infers invariants based on data-mining techniques [28], IODINE is a tool to detect invariants from hardware design [29], and Agitator is a commercial tool inspired by Daikon [30].

In terms of software security, invariants have been used as a measure of software integrity (data or code). Lorenzoli *et al.* proposed FFTV, a self-protecting technique that uses invariants to analyse and react to failures [31]. The technique infers likely invariants from the contexts that generate failures. Hence, whenever invariants from failing contexts are valid, FFTV applies a protection mechanism to prevent the failure either by correcting the execution issue or by avoiding the execution of the faulty code. Perkins *et al.* proposed ClearView which analyses a running system and uses Daikon to describe its behaviour in terms of likely invariants [32]. ClearView, in case of failures, automatically detects the failed invariants and proposes patches that re-establish the invariants' validity. Therefore, ClearView may protect from some attacks, such as code-injection.

Software attestation is a protection mechanism that aims at monitoring the integrity of a remote application without any hardware component to rely on to build root of trust, as presented by Sadeghi *et al.* [5]. Software attestation principles have been formalised by Coker *et al.* [33]. Recently, it is being taken into consideration specially for low-end devices in modern contexts such as Internet of Things [34, 35]. Even if the general architecture of software attestation is always the same, the element that can vary is the evidence used to monitor the target integrity. For instance, it can be a checksum of the binary or configuration files stored in the file system [36], a checksum of the binary loaded in memory [37], an application model [38], a measurement of the time spent to execute a particular piece of code [39]. Being a promising integrity measure, likely invariants have been introduced as integrity proof in software remote attestation contexts. Kil *et al.*, proposed ReDAS, a software attestation mechanism to monitor dynamic system properties [6]. ReDAS first extracts likely invariants from global variables values collected during system call invocations. Then, they claim the system is able to offer protection from tampering and limited protection from runtime memory corruption by evaluating invariants at runtime. Beliga *et al.*, proposed Gibraltar, a solution to detect kernel-level rootkits based on violation of inferred likely invariants [40]. Gibraltar, during an inference phase, observes data structures and values (e.g.,

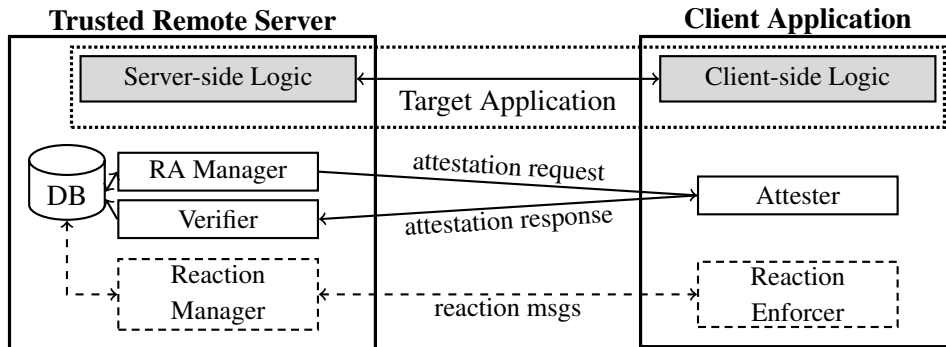


Figure 1: Architecture of our IM implementation.

entropy pool, processes list, page sizes) of the Linux kernel, thus it deduces invariants that are monitored during the detection phase. Wei *et al.* proposed an attestation mechanism that uses “scoped invariants”, likely invariants valid in specific scopes inside the program, to spot anomalies [41].

3 Remotely Monitoring Invariants

Figure 1 presents a general architecture of an IM system that relies on a trusted server to monitor the invariants of a target application running on untrustworthy client platform. This architecture inherits the design principles of past works in remote attestation [33] and integrates well with more general architectures that aim at applying more [42].

The target application may need server-side computations to perform its tasks (e.g., a client-server application). Therefore, the application to protect may be split in two parts, one Client-side Logic that interacts with one Server-side Logic (which is optional). These two parts are only concerned with the functionality/services the application intend to provide, they do not include any IM-related features that are performed by IM-specific server-side and client-side components.

A client-side *Attester* generates an *attestation response* to prove its integrity. That is, it collects and sends the value of (selected) variables available in memory (when the attestation response is generated). These attestations are sent to a trusted remote server where *Verifier* uses the received values to decree the integrity of the target application (see Section 8).

The reference architecture also includes a management component, the *Remote Attestation Manager*, that requires the target application to prove its integrity by sending *attestation requests* at non-deterministic time intervals. The Remote Attestation Manager may manage one or more instances of one or more target applications.

Our research aims at investigating the detective abilities of invariants, thus we did not consider the reactions. However, we mention that a server-side component, the *Reaction Manager*, is needed to decide, according to a policy³, when and how compromised applications need to be “punished”. Analogously, a client-side *Reaction Enforcer* may be useful to enforce the reactions that cannot be enforced on the server (e.g., graceful memory data corruption [43]).

Even if the general architecture is simple and well defined, verifying the integrity of a target application with likely-invariants and automating opens up a set of research and implementation issues we had to face when implementing this technique. First, we had to identify the likely-invariants of the target application, extract them, then we had to select the likely-invariants that could be useful for protection

³The policy may take decisions based on the type of the invariants failed and the frequency of failure as well user and business information, like the type of contract.

purposes. Once, likely-invariants are identified, we had to find methods to retrieve variables values from the client application. This scenario is made really complex, as values extracted from the memory of the application at client-side need to be recognized at server-side, i.e., bytes received from the client need to be correctly associated to variables and types by the Attester. Furthermore, since a protection technique is not useful if it cannot be easily applied, we had to investigate how to generalize the extraction of likely-invariants so that target applications can be protected without (or with limited) effort from the software developers. Moreover, we investigated how to instrument the target application to be protected with IM. That is, we researched how to add the Attester to the client application and how to prepare the server-side components to monitor a specific copy of the target applications.

We have thus developed our implementation of the IM technique, which is presented in Section 8. Moreover, we have implemented a tool chain that (semi-)automatically protects target applications with IM, which is presented in details in Section 9. Then, based on our implementations, we addressed another important set of questions, about the effectiveness of IM for protection purposes, related to the resource used by IM to attest target applications. Therefore, we measured the amount of management resources needed to monitor target applications protected with IM. We also measured how much the Attester affects the performance of the client application and determined if it affected the user experience with unacceptable overheads. Determining how the server-side components scale with the number of client applications to monitor was another interesting research question.

However, the most important research question concerned the possibility for a software attestation technique based on likely-invariants to effectively allows the identification of compromised application, which is extensively discussed in the next sections.

4 Daikon

Daikon is an excellent open source program that analyses traces (usually extracted with Kvasir for x86 platforms) to infer statistically significant likely-invariants. Nonetheless, since it is a tool designed for software engineering purposes, we identified some limitations that are worth analysing (and trying to overcome) in order to avoid issues when likely-invariants are used for protection purposes.

First, Daikon checks for invariants when procedures are entered and exited, likely invariants are thus functions' *pre-* and *post-conditions*. This is a feature for software maintenance purposes but it is a limitation from the protection point of view, as an attestation request may arrive in the mid of the procedure execution. Even worse, Daikon only considers global variables and variables passed to functions as parameters (the only ones Kvasir records in its traces) but it does not consider the inner scope local variables. In these cases, it is not clear which invariants to consider and it is not theoretically correct to use invariants, as variable may be in temporary states that violate invariants yet they are correct.

To reduce the impact of these limitations, we have implemented a source code manipulation phase (and added to our tool chain), during the instrumentation of the target application, which inserts dummy functions into every (original) function and into each inner scope⁴. In this way, we forced Kvasir to record variable values in order to obtain a more granular definition of invariants. Hence, we enabled the theoretically (more) sound verification of invariants also in the mid of a function. This approach could be beneficial for large functions when variables have a more complex life cycle. However, the impact of this improvement would need a more precise estimation. We note that improving the likely-invariants extraction tools can be an interesting investment for software protection developers, provided the major limitations of IM are actually overcome.

⁴This improvement follows the suggestions of the developers of Daikon and Kvasir for loop inner scopes <https://plse.cs.washington.edu/daikon/download/doc/daikon.html#Loop-invariants>

Furthermore, Daikon is unable to correctly manage data structures (e.g., the C `struct` constructs), and it only shows a minimum ability to treat mono-dimensional arrays. Furthermore, it deliberately ignores multidimensional data structures as they may contain numerical data that could be sources of false invariants. While some engineering is enough to overcome the limitation on data structures (e.g., expanding data structures in individual variables with some source code rewriting tool should work), limitations related to the impossibility to infer invariants on arrays are more difficult to overcome. Instrumenting the application to flatten multi-dimensional arrays may suffice to make Daikon processes multidimensional arrays. However, arrays and matrices may contain (numerical) data and inferring invariants on these values may increase the possibilities of false positives. Developers could annotate the multidimensional arrays for which they would like to compute invariants (e.g., in that case, they are not passed to Daikon's list of variables to exclude), to tell them from data structures that simply store input data. Although further studies on the impact of invariants computed on arrays and matrices may help in understanding when it could be advantageous, in our opinion (based on the use cases we have analysed), these structures do not contain data that are useful to decree the integrity of an application.

Daikon checks for 75 different *types of invariants*, furthermore, additional types of invariants can be specified by expert users with an internal language. Looking for a large number of invariants' types increases the chances that Daikon's output contains likely-invariants useful for IM purposes. However, it also increases the run time of the invariant detector algorithm. Nevertheless, in our opinion, Daikon list of invariants types considered by Daikon is big enough and it is also extensible.

In addition, to improve the often very frustrating performance of the tool extracting traces (Kvasir for x86 platforms), Daikon invariants' research scope can be limited to specific portions of the application code. At least theoretically, the "quality" of the extracted invariants should not be affected by limiting the research scope. From the research point of view, it is worth investigating if the scope restriction change the invariants detected. However, we did not consider worth investing in improving the trace extractor performance to avoid to resort to scope restriction.

Daikon last versions support abstract interpretation of variables' types, to avoid inferring invariants whose types are not compatible. Theoretically, enabling abstract interpretation should allow finding more effective invariants and thus reducing the likelihood of false positives. However, Daikon processing may last order of magnitudes more than with the basic algorithm. Indeed, we were not able to extract invariants in most of the applications we considered for use cases. Nonetheless, from our point of view, investing on improving this feature of Daikon was impossible.

Furthermore, Daikon neglects the execution history when computing invariants, it only considers variables' values divided by function. This means that it could lose some semantics, e.g., functions called from different points of the call graph may imply different likely-invariants. We have tried to improve this limitation by also considering Control Flow Graph information, as presented in Sections 9 and 5.5.

Finally, Kvasir is only able to extract traces to applications compiled without any optimization. Hence, the invariants extracted by Daikon are related to the variables allocated without optimizations. To keep the coherence, the whole protection process executed by our tool chain disables compiler optimizations to allow the \mathcal{A} to deterministically find the location of variables. Currently, there are no viable alternatives, however, future advancements will allow the reconstruction of the variables' location also with optimization.

All the identified limitations listed in this section are technological. They are not related to the way Daikon infers likely-invariants, they only concern the data passed to Daikon and how they are collected. A better likely-invariants extractor can be imagined and realised (with proper effort) and used to extract likely-invariants, but its performance and effectiveness would not change considerably the effectiveness of IM, as it will be more evident in Section 5.3. In other words, it is not Daikon, in our opinion, that may create problems to protections based on likely-invariants.

In Section 9, we have presented how we overcome the limitation in identifying invariants with variables from the inner scope by means of a *Function Injector*. This improvement has been considered during the assessment of IM. It is worth noting that all other technological limitations introduced by Daikon are either solved with easy engineering effort or not useful for protection purposes. Therefore, we did not invest resources for solving.

Trying to overcome all the limitations introduced by Daikon we have also analysed the state of the art in invariants extraction. However, we have not found recent interesting progresses in this field. One possible way to improve Daikon (or even substitute it) is the concolic analysis however, we did not find concrete application of this technique to invariants detection [44]. As a future work, we planned the investigation of applying concolic execution for software attestation.

5 On the effectiveness of likely-invariants for protection purposes

During the implementation of our IM protection and toolchain, we have experienced several limitations of IM. As anticipated, some depend of our implementation and the tools we have used. However, the two most important issues for the adoption of IM for protection purposes, are in our opinion the false positives and false negatives. Therefore, we first discuss them extensively.

False positives are limitations associated to likely-invariants that are independent of any extraction tool as they depend on the statistical methods used to infer invariants from the collected execution traces. Indeed, likely-invariants are true for the execution traces, not in general for any execution of the target application. In the case of IM, false positives are defined as the failure of invariants' verification also for unmodified target applications. We experienced that false positives are frequent in practice when likely-invariants are used. They are theoretically avoidable, with complete execution traces. If the traces passed to Daikon fully covered all the possible executions of the target application no false positive could manifest (i.e., likely-invariants would become true invariants). Since it is practically impossible to collect all the traces for all the possible inputs, we investigated how the number of likely-invariants varies with an increasing number of traces (there are too few true invariants). Therefore, we decided to investigate how increasing the number of traces and a larger number of input values affect the number and the quality of invariants and the likelihood of false positives (see Section 5.4).

On the other hand, *false negatives* are related to the impossibility of IM to detect some types of attacks. The fact that an attacker will modify invariants when tampering with an application so that IM will be helpful in discovering attacks is *the* assumption behind this technique. However, there are not proofs for this assumption. Consequently, false negatives are defined as the attacks against the application assets that are not causing the violation of any likely-invariants.

We have thus considered two scenarios to assess the impact of these issues for the use of IM to protect real applications in real contexts, that is, to allow IM to be used in the real world. The first scenario has been used to evaluate if IM is able to correctly identify original applications that have not been tampered with. Therefore, we have counted the number of false positives identified when running a target application that was not tampered with.

The second scenario has been used to evaluate the effectiveness of IM in identifying tampered applications. Therefore, we have estimated with theoretical, practical, and empirical methods if IM is able to successfully detect a set of attack paths against the target application.

We have selected and protected a set of target applications, which are our use cases. We have instrumented each of them with our tool chain and executed using two virtual machines with server-side and client-side components as in Figure 1. Then, for the second scenario, we have also implemented a set of attacks that compromised one or more assets in the target applications. The next sections present the use cases and how we validated IM for the two major limitations.

5.1 Use cases

We protected with IM the following open source applications written in C⁵:

- *Lynx*, a command line web browser⁶ that we have used to simulate Man-at-the-Browser scenarios;
- *MOC (Music On Console)*⁷ and *mpg123*⁸, two command line music players that we have used to simulate a DRM scenario;
- *oathtool*, an OTP generator and verifier⁹ that we have used to check IM effectiveness in high-sensitivity authentication software;
- *gamespace*, a test application for ncurses library¹⁰ that has been selected because it was properly dimensioned and interesting enough for an empirical assessment;
- *Bzip2*, a command line compression tool¹¹ that represents a category of utilities that are very pervasive in our lives for which the consequences of attacks have not been extensively explored.

5.2 Protecting the MOC player

The MOC Player is made of 91 C source files, 43478 LOC. MOC plays and pauses songs that can also be organized in playlists that are reproduced in linear or random order. We have modified the original MOC code to have two versions in order to simulate a typical DRM case. We assumed the original MOC application as the *Premium* version thus we created a *Free* version that restricts the functionalities available to the user. The Free version only allows playing playlists in random order (shuffle mode always on), and it does not allow skipping tracks or navigating the playlists (inspired by an online music streaming app). The changes have been implemented as `if` statements based on a preprocessor define, named `PREMIUM`:

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

Both the unused branch of the statements and the `PREMIUM` define are removed by the compiler and do not appear in the final binaries. Practically, we prevented the execution of the *next*, *previous* and *toggle shuffle* commands at user interface level, where the requested operations are forwarded to the proper player component. Additionally, we modified the player, the branch of the function that jumps to the next song in a playlist points to the random skip.

First, by analysing the call graph, we identified the functions to monitor, starting from the modified ones¹². We then protected the *Free* version with our tool chain, executed on an i7-4910MQ@2.90 GHz with 16 GB RAM. With small trace files (100 MB), MOC was protected in 5 h (including trace collection and likely-invariants inference). In another case, with 3 GB traces MOC protection took 50 h but we stopped Daikon's execution when only the 50% of traces was processed. Finally, with 1 GB traces we stopped MOC protection 140 h with Daikon at 1% computation. This suggests that not only the size, but

⁵The use cases information as well as the invariants inferred and statistical data are available at this URL <https://github.com/vucinic/im>

⁶<http://lynx.browser.org>

⁷<http://moc.daper.net>

⁸<http://www.mpg123.org>

⁹<http://www.nongnu.org/oath-toolkit/oathtool.1.html>

¹⁰<https://sourceforge.net/projects/game>

¹¹<http://www.bzip.org>

¹²Namely, 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`.

Target Application	Files	Functions	LOC	Likely-invariants.
<i>Lynx</i>	264	1890	193625	\emptyset
<i>Bzip2</i>	1	106	7010	5770 (193)
<i>MOC</i>	91	1215	43478	8553 (246)
<i>mpg123</i>	92	161	38060	7529 (3492)
<i>oathtool</i>	85	225	20469	3943 (2661)
<i>gamespace</i>	3	81	1752	29172 (18545)

Table 1: Statistical information on the use cases. \emptyset indicates that were not able to infer invariants with Daikon. Likely-invariants in parentheses are the non-duplicated ones.

also the type of traces may affect the performance. Furthermore, we cannot exclude bugs in the Daikon’s likely-invariants inference process.

We manually modified the original MOC makefile to adapt the Kvasir compilation parameters (added `-gdwarf-2`, substituted `-O2` with `-O0`). Daikon discovered 8553 invariants. We then manually analysed all the invariants and discovered that most of them were redundant because they were identically repeated pre- post-conditions or because they were only based on global variables thus always valid for each function and inner scope. In fact, MOC largely uses global variables whose value do not change during the execution (thus not interesting for monitoring purposes). We found only 246 distinct invariants, most of them related to global variables. Running Kvasir with the option `--ignore-globals` reduced the traces collected, the collection time, but Daikon only found 24 invariants.

Then, we ran Kvasir and Daikon with the Function Injector enabled. Unfortunately, Daikon was not able to infer additional invariants. We were expecting a limited number of new invariants, as MOC makes a limited use of inner scope variables, but none at all was surprising. We have carefully debugged our code to exclude our responsibility then we investigated the reason for this result. We discovered that Kvasir correctly reports in the traces the call to the injected functions but Daikon is unable to infer more invariants with the injector also when we reduced the threshold for accepting invariants. We experienced the same issue in all our use cases.

Analogous data about the other target applications are summarized in Table 1, which reports the number of source files, the LOCs of each use case, together with the invariants (total and distinct) discovered by Daikon with and without the injector enabled.

We analysed the performance of the protected MOC. At client-side, IM uses resources to compute the attestation response. The computation time of the attestation responses is the sum of the time to retrieve the values of the selected variables from the memory, the time to retrieve the ID of each variable from the VDS, assemble the response data, and the time to compute the hash.

In our experiments, the average time to compute an attestation was 442 ms, being the average number of variables in memory 87 and the average attestation data length 22591 B. Note that in our prototype we 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, we estimated the same impact on the other protected target applications. Since we expect target applications to be attested with an average frequency of a few minutes, we consider the delay introduced by IM actually manageable.

At server-side, \mathcal{R} uses resources to prepare an attestation request, however, it is only the time to generate a 256 bit nonce, while \mathcal{V} has to first compute an hash then use the variables ID to retrieve and check the invariants. In our experiments, the average 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,

however, in our opinion scalability may be granted with proper engineering of invariants indexing, use of data structures for fast verification, and, if really needed, dedicated hardware (e.g., FPGAs).

Then we simulated an attacker wanting to use Premium features on a Free app. We tested IM against two different attacks: 1) disable the shuffle mode to enable ordered playlist reproduction (by adding code that forces the value of the `shuffle` option in the `options` global variable used by the `go_to_another_file` function and code that bypasses the piece of code that randomises the playlist), and 2) attach a debugger to enable the `next` function (by trapping the call to `go_to_another_file` function and executing the (previously removed) code to jump to the next track in the playlist).

We ran the attacks and then we analysed 100 attestation responses for each attack. The first attack is always detected by IM: the alteration of the variables' value revealed the tampering (100% detection). On the other hand, the second attack is never detected by IM (100% false negatives). Indeed, the attack is mounted without altering variable values and it is outside the possibilities of IM. Note that if the attacker knows that the target application is protected with IM may choose other attack paths that are not recognizable (e.g., by cloning variables and working on clones, or by skipping with a debugger the playlist shuffling code).

To generalize the case of the music player, we repeated exactly the same process on `mpg123`. In this case, the functions to protect were `prepare_playlist`, `shuffle_playlist`, `get_next_file`, `playlist_jump`, `next_track`, `prev_track`

Daikon identified 8714 likely-invariants, most of them were duplicated (6434) because `mpg123` uses more global variables. Even in this case, the first attack was the only one detected.

5.3 Analysis of likely-invariants

In this section, we present a selection of likely-invariants extracted by Daikon on `Bzip2`. Being `Bzip2` a relatively small application, the entire protection process, including the likely-invariants extraction and instrumentation on an `i7-4910MQ@2.90 GHz` with 16 GB RAM, took 27 min and 71.5 min 23 s by enabling the abstract type interpretation. As anticipated, every invariant is associated to a precise point of the execution of the target application, i.e., when a function starts or just before the function is exited. For instance, for the `BZ2_bsInitWrite()` two groups of invariants have been computed, the ones listed under the following labels:

- `..BZ2_bsInitWrite()::ENTER`, which contains 458 invariants computed on the values available in memory at that time (input variables as well as global ones), and
- `..BZ2_bsInitWrite()::EXIT`, which contains 1187 invariants on the values available in memory at that time, several of them are also related to values of variables at functions enter (e.g., those using `orig` keyword).

We have manually analysed the `Bzip2` invariants to verify which ones could be used to tell compromised applications from untampered ones to improve the performance of our IM system. However, we have found several classes of invariants showing common properties that are worth presenting. We first present the following invariant:

```
::workFactor == 30
```

extracted from the `BZ2_bsInitWrite()` function, which allows us to introduce a first class of invariants. The `Bzip2` documentation reports that the `workFactor` variable stores an integer used to determine how to manage the cases highly repetitive input data that heavily affect the performance of the standard compression algorithm, and determine when to force the use of the fallback algorithm (which is 3 times slower). The default value of is 30 and, according to the `Bzip2` developer, gives reasonable behavior over

a wide range of circumstances. When computing our traces with Kvasir, we didn't change this input parameter, therefore, the invariant reports that the value of `workFactor` must always be 30. Therefore, legitimate users changing the default compression (e.g., with the `--exponential` option which assigns `workFactor=1`) would have been detected as attackers (e.g., a false positive). From this case we learned that the dependency on the inputs may generate false positives as the obtained invariants are *too restrictive*. A manual inspection (and the knowledge of the developers) may identify these invariants, remove them or try to consider more traces collected by changing the input parameters (see Section 5.4). Informally, let us consider the domain of an invariant, i.e., the Cartesian product of the domain of each variable involved in the invariant. 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.

Analogously, another class of invariants is not relevant from the protection point of view, as they are *specific of the protection environment*. For instance, they relate to the path of the input and output files we have used for collecting all traces:

```
::inName == "../inputs/in_big.txt"
::outName == "../inputs/in_big.txt.bz2"
s[].blockCRC == [4044990084]
```

Also in this case, a manual inspection would have led to the decision of excluding these invariants.

Another class of invariants, the *broad invariants*. They are very loosely limiting the values of variables to be of any use for software protection, that is, they almost always map to true. For instance, the following invariant states that the `incs` vector will contain at least one element:

```
::incs[] elements >= 1
```

Even if there are no reasons to exclude this invariant, users should be warned that they must not expect very much from its usefulness.

Moreover, some invariants belong to another class, the *inconsistent invariants*, which are built using variables that are not semantically related. For instance, we report invariants on arrays that use wrong variables as indices:

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

Analogously, the following likely-invariants are built on statistically relevant properties extracted from traces but have no semantic relevance and should be eliminated from the list of the invariants. As an example of likely-invariants that has not logical foundations from the `BZ2_bzCompress()::EXIT()` invariants domain:

```
::verbosity <= orig(action)
```

which reports that the verbosity level decided from the input (global variable) is less than or equal to the value of the action (i.e., compress and decompress) variable passed as input. Also in this case, a manual inspection may reduce unwanted false positives.

We have executed ten times the protected version of Bzip2 to estimate the false positives (on the original application). We have then evaluated all the non-redundant likely-invariants, not only the manually selected ones. We forced \mathcal{R} to ask for attestations more often than usually required, that is, every 30 s. During the simulation 17 attestation responses were received. Only 10 out of 17 requests included variable values. In the remaining seven cases, the application received the attestation request and was interrupted to compute the attestation in a (low-sensitivity) function for which no likely-invariants were defined. Therefore, we report that it is perfectly legitimate, as expected, that the target application answers with an empty attestation response. This means that also an attacker may use the same approach to bypass IM protection with the same strategy.

With the data from the 10 attestation responses, the system evaluated 165 out of 179 likely-invariants. The verification of 19 invariants failed, thus generating false positives. Eight false positives were due to the use of `workfactor = 1` command input. However, we also recorded 11 additional false positives independent of the work factor value, some related to file names and some other to internal management variables, like the `deleteOutputOnInterrupt` variable.

5.4 Analysis of invariants depending on the extraction strategies

We have noted that some false positives could have been avoided with an exhaustive (or at least larger) set of execution traces, therefore we have arranged another analysis, which is presented here for another security related application: `oathtool`. `Oathtool` generates and validates event-based HMAC OTPs against a key and a window of computation iterations. The tool is also able to manage time-based OTPs but we did not consider that capability for our considerations. An event-based HOTP is deterministic, as it only depends on the given inputs and the internal constant values. We extracted invariants from the validation phase: the application takes as input the OTP to validate, the key and the iterations window in which the verification has to be performed.

Single execution traces The first extraction we performed has been based on a single execution of the application, it validates a previously generated HOTP. We obtained 1580 invariants, 892 related to global variables and 688 related only to local variables (function parameters). By analysing the obtained invariants, we noticed two main facts. First, most of the invariants are meaningless and do not describe any security or behavioral property of the application, in addition they are duplicated for several program points, for instance:

```
::program_name < ::gengetopt_args_info_usage
```

Second, invariants that may be considered meaningful at a first glance, are too restrictive. That is, we found invariants that describe application depending on the input but they are too specific for that values to properly describe valid application:

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

It is worth noting that in our simulations specific invariants lead to false positives, as they are too tightly bound to a subset of the inputs and detect as compromised perfectly valid target applications whose only fault was being executed with different inputs.

Multiple execution traces. In order to overcome the restriction introduced by the limited execution we tried to extract invariants from multiple executions. In particular, we executed the HOTP validation 100 times in two different ways: by passing always the same value to validate and by passing a different value to each execution. To collect traces, we always used valid OTP values to be validated. The first one (single value) led to gather 2643 invariants (1823 involving globals, 820 involving only local variables). From the second one (diversified inputs) we obtained 2660 invariants (1838 involving globals, 822 involving only local variables).

Therefore, we expected to obtain more descriptive invariants and to limit the amount of useless invariants. However, the collected evidences have not confirmed our expectations. Indeed, the overall number of collected invariants increased but the additional invariants (we have individually evaluated all of them), are not useful and do not describe any interesting property of the application. In addition, the invariants that we observed in the single execution as potentially useful became too broad to be of any use (e.g., `moving_factor >= 0`) or simply disappeared (e.g., none of the invariants involves the `otp` variable).

Merging traces. In alternative, instead of computing invariants on all the traces, we tried to first extract invariants from single executions and then to merge the inferred invariants. We noticed in this case that several invariants extracted from different executions were conflicting, e.g., two invariants on the same array stated that it contained exactly one value and more than one value. In this case, all the conflicting invariants should be ignored, unless a manual inspection suggests a better strategy (e.g., merge them in a broad invariant).

To summarize, the analysed target application is deterministic and its execution is supposed to directly depend on the inputs. Thus, we were expecting to be able to identify 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 behavior of the application. To try to better describe the application we used multiple execution traces with different input values. Nevertheless, this strategy makes the specific invariants disappear, instead, the extractor infers broad invariants that are useless for integrity checking purposes. Even worse, if invariants collected from different executions are merged a posteriori, there is the risk to find conflicting predicates.

5.5 Relating likely-invariants with control flow graph

To better characterize invariants and to limit the chances to broad invariants, we have tried to relate invariants to execution points taken from the Control Flow Graph (CFG) of the target application. The aim was to add more dynamic information to IM. The research question we wanted to answer concerned the possibility that the likely-invariants computed on pre- and post-conditions of the same function are different if the function is called from different points of the CFG?

To answer this question we have post-processed with a script the traces extracted with Kvasir in order to obtain, for each function that is called from different points of the CFG, one independent trace file for each call point. For instance, from MOC use case we have obtained two independent traces associated to the `audio_play` and three associated to `play_it` functions, and passed to Daikon to process them. We avoided to generate independent traces for `options_get_int` as it was called by 106 functions.

By passing Daikon the independent trace files we have obtained different sets of invariants associated to the “traces split function”. Unfortunately, we have not noticed significant improvements in the quality of invariants, as broad invariants changed but remained broad. It is worth noting that the absence of notable differences does not mean that post-processing traces does not improve likely-invariants quality in general. In our opinion, the lack of differences is better explained by the fact that the applications we considered have a rather simple Control Flow Graph. Nevertheless, we report that this improvement may fail in practice.

5.6 Variables in memory

As already anticipated in Section 5.3, by running our experiments, we discovered another limitation of IM. As said before, the inferred invariants are pre- post-conditions associated to functions, hence, they are only evaluable when the associated functions are executed, even for invariants only based on global variables. By chance, we have selected applications that are in the idle state most of the time waiting for user input. During the idle period, very few functions are being executed, often just the main (e.g., the library functions that play tracks), thus few variables are available in memory. Hence, the risk is that either IM evaluates almost always the same invariants, or checks no invariants at all, if the idle functions are not monitored.

6 Empirical assessment

In order to further support our analysis of the detection ability of IM we have used data from an experiment involving master students from Politecnico di Torino. The experiment has been performed in the context of the ASPIRE project and aimed at checking the ability of the client-server code splitting technique [45]. Students were asked to attack the *gamespace* application, a testing application for the ncurses library, that shows the user an interface where a set of entities are moving. In that scenario, the user is allowed to move one piece, with respect to hurdles, in the preferred direction. Each prompted direction make the piece move 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 interface¹³. Each graphical element, namely a `Player`, is represented as a list of `Point` elements that specify the position of each point of the figure to draw. When an element has to be moved, 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.

The attack that the participants were asked to perform consisted of forcing the application to move of two steps at each key press 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.

From the collected reports we found that successful attacks have exploited a small set of changes to the application. Then, we analysed the ported changes in order to understand if the IM technique could be able to identify them.

The involved students mainly followed two statistically balanced strategies: modify data and modify control flow. For the first strategy, they altered the content of `point->row` and `point->col` content before the call to moving functions.

This kind of attack is supposed to be identifiable by IM, in fact it alters the content of variables. However, it was not detected. Our investigation reported that it happened because of broad invariants. Indeed, the value of variables `point->col` is in the integer range $[1, 80]$ and `point->row` is in $[1, 25]$, which is the actual movement area. By doubling the movement (yet maintaining the point in the movement area), no invariants are violated. We also report that surprisingly Daikon is not able to identify the boundaries of the movement area. 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 an interesting 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 evidences.

7 Discussion

From our experience, we can state that technological limitations do not really affect the effectiveness of this technique. Indeed, they can be (completely or partly) addressed to make IM a protection technique that can work in practice, provided IM is associated to techniques that avoid attaching debuggers [46]. Indeed, there is nothing to do when attacks can be mounted by attaching a debugger without altering variables values.

However, as anticipated, false positives and negatives do affect the possibility of using IM for protection purposes. From our analyses, we have noted that false positives are very likely and may lead to tamper countermeasures to be triggered even if the target application is correct. The number of false

¹³`movePoint, moveSolidPlayer, movePlayer, movePointInMap, moveFlexiblePlayer`

positives may scare developers wanting to protect their applications with IM because of the management costs. Indeed, from our experiments on simple applications, we expect that reducing them to a physiological level and implementing proper reaction policies that are able to live with false positives may be very fatiguing and frustrating. The learning phases, which should be done by developers prior to 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 to us with broad invariants. After all these considerations, developers wanting to protect their target applications with IM are warned that likely-invariants may be practically unusable.

Nonetheless, false negatives are the most important and dangerous issue for a technique aiming at attesting the integrity of target applications. False negatives are a fundamental limitation to the use of likely-invariants for protection purposes. Indeed, we have experienced them in all use cases we have analysed several cases of false negatives (see Section 5). From the analysis of thousands of likely-invariants and the manual inspection of a huge amount of logs produced by the Verifier of our IM system, we are convinced that false negatives are due to the lack of a precise link between attacks that compromise the assets in the target application and the likely-invariants extracted, which is the baseline assumption for using likely-invariants for protection purposes. Although it has been noticed on target applications protected with our IM, in our opinion, this is a general limitation of likely-invariants. All the trace collection and likely-invariants extraction is a statistical process that never considers the semantic, that is, the nature of assets, the security properties that developers want satisfied on them (i.e., the integrity), and attacks paths and strategies aiming at compromising them. In other words, there is not a clear way to assume that an attack compromising one or more assets of the the target application will also violate likely-invariants.

Establishing a clear inference among violation of invariants and attacks that actually compromise application assets is a research issue of uttermost importance for whoever may be interested in developing (automatic) techniques to protect target applications by monitoring likely-invariants. It could be worth investigating the use of data dependency and data flow analysis techniques to improve the semantic of the inferred statements. Moreover, IM must evolve to integrate with techniques aiming at modifying and reconfiguring applications at runtime (e.g., code mobility). Combining these techniques could be a valuable direction for improvements and practical application of IM. Empirical assessment may also identify human strategies that circumvent IM and analysis of empirical data may lead the design of improved versions of IM. Until the research has not produced advancements that may permit to override our negative results, protection developers are warned against investing on IM.

8 Detailed description of our prototype

Our implementation of the IM technique matches the architecture in Figure 1. It uses, an Attester (\mathcal{A}) at client side, and a Verifier (\mathcal{V}) and Remote Attestation Manager (\mathcal{R}) at server side. We assume that, when the target application is launched, it notifies the trusted server, regardless of the fact that it may need the execution of some Server-side Logic. Therefore, the trusted remote server knows that the target application is up and running, thus \mathcal{R} adds the application in the list of the ones that will be asked for integrity proofs according to the process reported in Figure 2. We also assume that client and server perform a mutual authentication phase and, strongly suggested, negotiate a secure channel where to transmit attestation requests and replies.

The Remote Attestation Manager starts an attestation transaction process that consists in following phases, as depicted in Figure 3:

- \mathcal{R} wakes up and selects the target application that has to prove its integrity,


```

Data: scheduling_list, serving_thread_pool
begin
  scheduling_list  $\leftarrow$   $\emptyset$ ;
  for  $t \in$  serving_thread_pool do
    | initialise  $t$  with servingRoutine;
  end
  while True do
    | wait connection from  $\mathcal{A}$ ;
    | read client parameters from DB;
    | broadcast new_client event to thread_pool;
  end
end

```

Figure 2: Remote Attestation Manager algorithm.

```

Data: scheduling_list,
       wake_reason  $\in$  {timeout, new_client}
begin
  wake_reason  $\leftarrow$  timeout;
  while True do
    | client  $\leftarrow$  first of scheduling_list;
    | time_to_sleep  $\leftarrow$  time of client;
    | timeout-wait for new_client event;
    | if wake_reason = timeout then
      | | nonce  $\leftarrow$  random bytes;
      | | store nonce into requests table in DB;
      | | send nonce to scheduled client;
      | | insert client into scheduling_list;
    | else
      | | insert new_client into scheduling_list;
    | end
  end
end

```

Figure 3: Serving Routine algorithm.

- \mathcal{R} prepares an attestation request for the selected target application,
- \mathcal{R} stores all data to a local DB to make it available for \mathcal{V} ,
- \mathcal{R} schedules the next attestation request for that specific client, finalises all the transaction data in the local DB (and potentially triggers the Reaction Manager), then
- \mathcal{R} enters again the sleep mode.

\mathcal{R} decides when the application will be next attested by selecting a random time interval that keeps constant the average time between two attestations, a parameter that can be set up individually for each instance of the target application.

An attestation request is a message that only conveys a random 256 bit nonce n . The *Attester* (\mathcal{A}) processes the attestation requests. It collects the values of selected variables among the ones available in memory at the moment of the attestation request receiving. Variables may be in the stack, in the data segment, in a register, in a known location in memory, in a memory location referenced by a register, or they can be translated as a constant value. Variables are uniquely identified during the instrumentation phase to allow \mathcal{V} to unambiguously recognise them. Unique IDs and DWARF information are used to build a Variables Data Structure (VDS) that is injected into the client during the instrumentation phase. This data structure is used by \mathcal{A} to send the server information needed to identify the sent variables. In our implementation, the target application contains the VDS as a whole blob. Information about monitored variables is sensitive for an attacker trying to compromise IM but the variable identification and location data need to be made available at the client, as we did not find an effective solution to avoid the *injection of the VDS*. Certainly, there are more stealthy ways to inject the VDS in the application than just putting an entire data structure but, we did not invest resources in engineering a way to hide the VDS for our prototype.

In the first implementation of our prototype, \mathcal{A} sent all the variables in memory. This simplified approach suffers of a serious drawback: depending on the target application, the attestation responses may be too large in size. Indeed, when the target application has too many global variables (which are always accessible thus always sent) or when it uses big buffers or big data structures, attestation responses may become unmanageable. We have addressed this technological limitation by developing a variant of the Attester that only sends a subset of the variables based on the value of the nonce. In short,

all the invariants are associated to progressive unique integer identifiers. However, only the ones that are in a precise relation with the nonce n are considered when computing attestations. Invariants (and related variables) are selected if their identifier satisfy the following condition:

Formally, invariants are associated to the variables they use by means of the function v :

$$\begin{aligned} v: I &\rightarrow 2^V \\ i_k &\mapsto \{v_{k,1}, v_{k,2}, \dots\} \subseteq V \end{aligned}$$

where V is the set of all the variables defined in the target application and I is the set of all the invariants. As an helper, we define the function that associates a set of invariants to the set of the variables used by at least one of the input invariants as:

$$\begin{aligned} v^I: 2^I &\rightarrow 2^V \\ i_{kk} &\mapsto \bigcup_{i_k} v(i_k) \subseteq V \end{aligned}$$

Invariants associated to the nonce n are determined with the μ function:

$$\begin{aligned} \mu: \mathbb{N} &\rightarrow 2^I \\ n &\mapsto \{i_{n,1}, i_{n,2}, \dots\} \subseteq I \end{aligned}$$

Therefore, \mathcal{A} only sends to the server variables whose IDs are associated to the invariants related to the nonce n , that is, the variables to send V_s are determined as:

$$V_s = (v^I \circ \mu)(n)$$

Then, invariants to be evaluated are randomly selected according to nonce randomness.

It is worth noting that this approach reduces the data sent by the \mathcal{A} but it may also reduce the number of invariants that can be verified. Hence, it may reduce the effectiveness of the detection ability of IM and requires additional effort for the validation. In addition, this approach reduces the chance to evaluate invariants. It could happen that the variables selected for extraction are not available in memory, hence all the related invariants could not be evaluated.

Independently from the selected invariants, the Attester works as outlined in Figure 4; that is it: 1) stops the execution of the target application; 2) based on the VDS, unwinds the call stack and, for each stack frame, reads the instruction pointer and deduces the associated function; 3) for each deduced function and depending on the instruction pointer, identifies and locates the extractable variables; 4) collects the variables values according to the nonce.

Finally, \mathcal{A} assembles the data d to send:

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

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 and $\text{Value}(v_i)$ is the value of the variable found in memory for each variable in V_2 . Then, it sends \mathcal{V} the attestation:

$$a = d || H(d || n || appID || S)$$

where H is a hash function of choice (we support SHA1, SHA256, and Blake2), $appID$ are optional data that unequivocally identifies the running application (see Section 9), n is the nonce, and S are optional data that relate the client to the server (e.g., secrets shared during the mutual authentication).

\mathcal{A} needs to access the same memory areas as the original application, therefore, it *cannot be an external entity* (e.g., another process or program). Therefore, \mathcal{A} 's code is injected in the client application. To ease the implementation (i.e., avoid source code modifications), \mathcal{A} is launched before the

main, thus immediately recognizable and easy to disable. Indeed, \mathcal{A} initialisation function is given the constructor attribute (which is valid both for gcc and llvm) and tells the compiler to insert the function invocation before the main function call or before the dynamic library is loaded. This limitation can be overcome by properly engineering the client application. However, we decided not to invest more resources on this technological limitation. Indeed, solving it would have required a lot of effort and produced research results of limited impact. Getting variables values with an *external* \mathcal{A} requires a process working at higher level of privileges (e.g., with VM introspection). Practically, it looks feasible even if this introduces other security and privacy issues (may it monitor the whole client?) and technological ones (how easy is to identify and stop this external Attester?).

```

Data:  $a, d, appID, S, vars\_list, n$ 
begin
  parse VDS from memory;
  set all parsed var to be extracted;
  connect to  $\mathcal{A}$ ;
  save connection parameters into  $S$ ;
  wait attestation request;
   $nonce$  from request;
   $d \leftarrow \emptyset$ ;
   $n \leftarrow 0$ ;
  append  $nonce$  ID to  $d$ ;
  if restrict invariants version then
    | set variables to extract;
  end
  foreach  $var \in selected\ vars$  do
    | if  $var$  is available in memory then
      | |  $n \leftarrow n + 1$ ;
      | | append  $(v_{ID}(var), value(var))$  to
      | | |  $vars\_list$ ;
    | end
  end
   $d \leftarrow n || vars\_list$ ;
   $a \leftarrow d || H(d || N || appID || S)$ ;
  send  $a$  to  $\mathcal{V}$ ;
end

```

Figure 4: Attester algorithm.

```

Data:  $a, appID, vars, evaluable\_invariants$ 
while True do
  wait connection from  $\mathcal{A}$ ;
  receive attestation data  $a$ ;
  verify attestation data hash;
  if hash not verified then
    | record the event in the DB;
  else
    foreach  $(v_{ID}(var), value(var)) \in a$  do
      | insert  $(v_{ID}(var), value(var))$  into  $vars$ ;
    end
    foreach  $f_{inv}(V)$  in DB do
      | if  $\forall v \in V \implies v \in vars$  then
        | | insert  $f_{inv}(v_i)$  into
        | | |  $evaluable\_invariants$ ;
      | end
    end
    foreach  $f_{inv}(v_i)$  in  $evaluable\_invariants$  do
      | foreach  $v_{ID}(x) \in f_{inv}(v_i)$  do
        | | replace  $v_{ID}(var)$  with  $value(var)$ ;
      | end
      |  $f_{inv} \leftarrow f_{inv}(v_i)$ ;
      | record  $f_{inv}$  result in DB;
    end
  end
end

```

Figure 5: Verifier algorithm.

The *Verifier* (\mathcal{V}) uses the attestation response a received from \mathcal{A} to decrease the integrity of the application, as detailed in Figure 5. \mathcal{V} first checks the correctness of the received hash. It reconstructs the variable values from the responses, use them to evaluate invariants, and logs into the DB all the evaluation results, both positives and negatives. That is, \mathcal{V} uses the $v_{ID}(v_i)$ in a to determine the invariants that can be verified (which are the ones for which all the variables v_{ID} are in a), then employs the values $Value(v_i)$ to actually evaluate the invariants. Formally, the invariants that can be evaluated are determined with this formula

$$i_k \text{ evaluated} \iff v(i_k) \subseteq V_s$$

where, at server side, V_s is assembled by collecting all the $v_{ID}(v_i)$ received in the attestation.

The verification of the invariants must be done with variable values taken from memory at the same time, that is, from a single attestation response. Using values from different attestations responses may invalidate invariants even if the target application is not compromised. For instance, given the code snippet below:

```

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

```


to functions, can also consider inner scopes variables. For instance, for this function:

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

the Injector finds the `c` and `a` variables and generates the function:

```
void _____injectedFunction_rand (int c, double d){}
```

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

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

This component also outputs a description file where it links the injected functions to the function where they have been injected, to allow reconstructing each invariant from the output Daikon will produce in a later step.

The output of the Injector is then processed with the **standard compiler** to generate a binary that can be traced by Kvasir¹⁴. The compiler is run with debugging options enabled (i.e., `-g -gdwarf-2`) and optimisation options disabled (i.e., `-O0`). Then, the **Invariants Extractor** launches Kvasir to collect execution traces and, subsequently, it calls Daikon to analyse the traces. Lacking a standard way to compile all the C applications, the *compiler options for Daikon* and for its trace collection helpers as well as the ones to compile the target application need to explicitly passed as input to our tool chain (we derived them for our use cases from the makefiles). The Invariants Extractor produces two outcomes: the invariants detected by Daikon that serve for monitoring purposes, which can be (optionally even if strongly suggested) manually filtered by users, and the list of the variables to retrieve at runtime to verify the selected invariants.

The second process also starts from the unprotected application sources. The application, together with the Attester files, is compiled by standard compiler. Starting from the obtained binary, the **DWARF Parser** analyses the DWARF symbols and collects the information (location descriptions) needed to retrieve variables values at run-time. Furthermore, it assigns a unique ID to all the variables to monitor, and stores all these data in the DB (*variables formalisation*). For the first version of the prototype, we used a progressive integer, whereas for the second version of the prototype (that only sends a subset of the variables in memory), variables' identifiers are generated so that from invariants identifiers it is possible to derive the set (or more precisely a superset) of the variables that are needed to verify it. The parser also produces the VDS as a binary file.

Finally, an **Invariant Interpreter** uses the output of the Invariants Extractor and variables' info to optimise the invariants representation for fast verification and obtain the *invariants formalisation* then stored in the DB.

In the end, the application and the Attester are compiled without debugging symbols. Then the VDS is injected in the resulting binary. This step completes the generation of the protected application.

¹⁴Note that Kvasir works for x86 architectures, unfortunately, hence, for other architectures we were not able to use Daikon.

10 Conclusions

This paper has presented the Invariants Monitoring protection technique, which aims at remotely monitoring the integrity of a running application by checking likely-invariants. We have analysed the literature on invariants and presented the foundations of the invariants monitoring technique, our implementation of IM, and the tool chain we have used to automatically protect an arbitrary application. We have also analysed all the found limitations, to identify the research and technological efforts needed to overcome them. Our results have highlighted that this protection technique suffers from false positives and false negatives. False positive are very likely and may seriously affect the management cost. False negatives are also very likely and we proved with use cases and data from an empirical assessment that IM is unable to detect a large set of attacks.

We have also identified some research lines that may resurrect this software attestation technique, even if our experience suggests it will be very unlikely.

Acknowledgments

The research described in this paper is part of the ASPIRE project, co-funded by the European Commission (FP7 grant agreement no. 609734).

References

- [1] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, “A security framework for the analysis and design of software attestation,” in *Proc. of the 2013 ACM SIGSAC conf. on Computer and Communications Security (CCS’13), Berlin, Germany*. ACM, November 2013, pp. 1–12.
- [2] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, “Design and implementation of a TCG-based integrity measurement architecture,” in *Proc. of the 13th Conference on USENIX Security Symposium (SSYM’04), Berkeley, California, USA*, vol. 13. USENIX Association, August 2004, pp. 223–238.
- [3] T. Committee *et al.*, “Trusted computing platform alliance (TCPA) main specification v1,” 1b. Technical report, TCPA Alliance, Tech. Rep., 2002.
- [4] F. Valenza, T. Su, S. Spinoso, A. Lioy, R. Sisto, and M. Vallini, “A formal approach for network security policy validation,” *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, vol. 8, no. 1, pp. 79–100, 2017.
- [5] A.-R. Sadeghi and C. Stübke, “Property-based attestation for computing platforms: caring about properties, not mechanisms,” in *Proc. of the 2004 workshop on New security paradigms (NSPW’04), Nova Scotia, Canada*. ACM, September 2004, pp. 67–77.
- [6] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, “Remote attestation to dynamic system properties: Towards providing complete system integrity evidence,” in *Proc. of the 2009 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN’09), Lisbon, Portugal*. IEEE, June 2009, pp. 115–124.
- [7] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, October 1969.
- [8] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.
- [9] J.-M. Jazequel and B. Meyer, “Design by contract: the lessons of ariane,” *Computer*, vol. 30, no. 1, pp. 129–130, January 1997.
- [10] F. Cristian, “Exception handling and software fault tolerance,” *IEEE Transactions on Computers*, vol. 31, no. 6, pp. 531–540, June 1982.
- [11] S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in *Proc. of the 24th International Conference on Software Engineering, Orlando, Florida, USA*. IEEE, May 2002, pp. 291–301.

- [12] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, February 2001.
- [13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The daikon system for dynamic detection of likely invariants,” *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, December 2007.
- [14] A. Viticchié, C. Basile, and A. Lioy, “Remotely assessing integrity of software applications by monitoring invariants: Present limitations and future directions,” in *Proc. of the 12th International Conference on Risks and Security of Internet and Systems (CRISIS’17), Dinard, France*, ser. Lecture Notes in Computer Science, vol. 10694. Springer, Cham, September 2017, pp. 66–82.
- [15] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, “VCC: A practical system for verifying concurrent C,” in *Proc. of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs’09), Munich, Germany*, vol. 5674. Springer, Berlin, Heidelberg, August 2009, pp. 23–42.
- [16] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, “Path invariants,” *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 300–309, June 2007.
- [17] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “A static analyzer for large safety-critical software,” in *Proc. of the ACM SIGPLAN 2003 Conference on Programming language design and implementation (PLDI’03), San Diego, California, USA*. ACM, June 2003, pp. 196–207.
- [18] G. Klein, M. Norrish, T. Sewell, H. Tuch, S. Winwood, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, and R. Kolanski, “seL4: Formal verification of an OS kernel,” in *Proc. of the 22nd ACM SIGOPS symposium on Operating systems principles (SOSP’09), Big Sky, MO, USA*. ACM, October 2009, pp. 207–220.
- [19] N. Delgado, A. Gates, and S. Roach, “A taxonomy and catalog of runtime software-fault monitoring tools,” *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, December 2004.
- [20] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst, “Dynamic inference of abstract types,” in *Proc. of the 2006 International Symposium on Software Testing and Analysis (ISSTA’06), Portland, Maine, USA*. ACM, July 2006, pp. 255–265.
- [21] T. Xie and D. Notkin, “Tool-assisted unit test selection based on operational violations,” in *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE’03), Montreal, Quebec, Canada*. IEEE, October 2003, pp. 40–48.
- [22] C. Csallner, Y. Smaragdakis, and T. Xie, “DSD-Crasher: A hybrid analysis tool for bug finding,” *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 2, pp. 1–37, April 2008.
- [23] D. Schuler, V. Dallmeier, and A. Zeller, “Efficient mutation testing by checking invariant violations,” in *Proc. of the 8th International Symposium on Software Testing and Analysis (ISSTA’09), Chicago, Illinois, USA*. ACM, July 2009, pp. 69–80.
- [24] T. W. Schiller and M. D. Ernst, “Reducing the barriers to writing verified specifications,” *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 95–112, November 2012.
- [25] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve, “Using likely invariants for automated software fault localization,” *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 139–152, April 2013.
- [26] C. Lemieux, D. Park, and I. Beschastnikh, “General ltl specification mining (t),” in *Proc. of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE’15), Lincoln, Nebraska, USA*. IEEE, November 2015, pp. 81–92.
- [27] C. Csallner, N. Tillmann, and Y. Smaragdakis, “DySy: dynamic symbolic execution for invariant inference,” in *Proc. of the 30th ACM/IEEE International Conference on Software Engineering (ICSE’08), Leipzig, Germany*. ACM, May 2008, pp. 281–290.
- [28] C. Ackermann, R. Cleaveland, S. Huang, A. Ray, C. Shelton, and E. Latronico, “Automatic requirement extraction from test cases,” in *Proc. of the 1st International Conference on Runtime Verification (RV’10), St. Julians, Malta*, ser. Lecture Notes in Computer Science, vol. 6418. Springer, November 2010, pp. 1–15.
- [29] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, “Iodine: a tool to automatically infer dynamic

- invariants for hardware designs,” in *Proc. of the 42nd ACM Design Automation Conference (DAC’05)*, Anaheim, California, USA. IEEE, September 2005, pp. 775–778.
- [30] M. Boshernitsan, R. Doong, and A. Savoia, “From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing,” in *Proc. of the 2006 ACM International Symposium on Software Testing and Analysis (ISSTA’06)*, Portland, Maine, USA. ACM, July 2006, pp. 169–180.
- [31] D. Lorenzoli, L. Mariani, and M. Pezze, “Towards self-protecting enterprise applications,” in *Proc. of the 18th IEEE International Symposium on Software Reliability (ISSRE’07)*, Trollhattan, Sweden. IEEE, November 2007, pp. 39–48.
- [32] J. H. Perkins, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, M. Rinard, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, and S. Sidiroglou, “Automatically patching errors in deployed software,” in *Proc. of the 22nd ACM SIGOPS symposium on Operating systems principles (SOSP’09)*, Big Sky, Missouri, USA. ACM, October 2009, pp. 87–102.
- [33] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O’Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, “Principles of remote attestation,” *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, April 2011.
- [34] J.-W. Ho, “Distributed software-attestation defense against sensor worm propagation,” *Journal of Sensors*, vol. 2015, pp. 1–6, March 2015.
- [35] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, “C-FLAT: control-flow attestation for embedded systems software,” in *Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security (CCS’16)*, Vienna, Austria. ACM, October 2016, pp. 743–754.
- [36] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, “Terra: A virtual machine-based platform for trusted computing,” *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 193–206, December 2003.
- [37] A. Viticchié, C. Basile, A. Avancini, M. Ceccato, B. Abrath, and B. Coppens, “Reactive attestation: Automatic detection and reaction to software tampering attacks,” in *Proc. of the 2016 ACM Workshop on Software PROtection (SPRO’16)*, Vienna, Austria. ACM, October 2016, pp. 73–84.
- [38] T. Jaeger, R. Sailer, and U. Shankar, “PRIMA: policy-reduced integrity measurement architecture,” in *Proc. of the 11th ACM symposium on Access control models and technologies (SACMAT’06)*, Tahoe City, California, USA. ACM, June 2006, pp. 19–28.
- [39] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, “Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems,” *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 1–16, October 2005.
- [40] A. Baliga, V. Ganapathy, and L. Iftode, “Detecting kernel-level rootkits using data structure invariants,” *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 5, pp. 670–684, September 2011.
- [41] J. Wei, C. Pu, C. V. Rozas, A. Rajan, and F. Zhu, “Modeling the runtime integrity of cloud servers: a scoped invariant perspective,” in *Privacy and Security for Cloud Computing*, ser. Computer Communications and Networks. Springer, London, 2013, pp. 211–232.
- [42] B. De Sutter, P. Falcarin, B. Wyseur, C. Basile, M. Ceccato, J. d’Annville, and M. Zunke, “A reference architecture for software protection,” in *Proc. of the 13th Working IEEE/IFIP Conf. on Software Architecture (WICSA’16)*, Venice, Italy. IEEE, April 2016, pp. 291–294.
- [43] G. Tan, Y. Chen, and M. H. Jakubowski, “Delayed and controlled failures in tamper-resistant software,” in *Proc. of the 8th Workshop on Information Hiding (IH’06)*, Alexandria, Virginia, USA, ser. Lecture Notes in Computer Science, vol. 4437. Springer, Berlin, Heidelberg, July 2006, pp. 216–231.
- [44] R. Baldoni, E. Coppa, D. C. D’Elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Computing Survey*, vol. 51, no. 3, June 2018.
- [45] M. Ceccato, M. D. Preda, J. Nagra, C. S. Collberg, and P. Tonella, “Trading-off security and performance in barrier slicing for remote software entrusting,” *Automated Software Engineering*, vol. 16, no. 2, pp. 235–261, February 2009.
- [46] B. Abrath, B. Coppens, S. Volckaert, J. Wijnant, and B. D. Sutter, “Tightly-coupled self-debugging software protection,” in *Proc. of the 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW’16)*, Los Angeles, California, USA. ACM Press, December 2016, pp. 7:1–7:10.

Author Biography



Alessio Viticchié received the M.Sc. degree in Computer Engineering in 2015 from the Politecnico di Torino, where he is currently a research assistant and a Ph.D. student. His research interests are concerned with software security, software attestation, and software protection techniques design and assessment.



Cataldo Basile received a M.Sc. (summa cum laude) in 2001 and a Ph.D. in Computer Engineering in 2005 from the Politecnico di Torino, where he is currently a research associate. His research is concerned with policy-based management of security in networked environments, policy refinement, general models for detection, resolution and reconciliation of specification conflicts, and software security.



Fulvio Valenza received the M.Sc. (summa cum laude) in 2013 and the Ph.D. (summa cum laude) in Computer Engineering in 2017 from the Politecnico di Torino. His research activity focusses on network security policies. Currently he is a Researcher at the CNR-IEIIT in Torino, Italy, where he works on orchestration and management of network security functions in the context of SDN/NFV-based networks and industrial systems.



Antonio Lioy is Full Professor at the Politecnico di Torino, where he leads the TORSEC cybersecurity research group. His research interests include network security, policy-based system protection, trusted computing, and electronic identity. Lioy received a M.Sc. in Electronic Engineering (summa cum laude) and a Ph.D. in Computer Engineering, both from the Politecnico di Torino.