Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks

Alessio Viticchié,
Cataldo Basile
Politecnico di Torino
Torino, Italy
{alessio.viticchie,
cataldo.basile}@polito.it

Andrea Avancini,
Mariano Ceccato
Fondazione Bruno Kessler
Trento, Italy
{anavancini,ceccato}
@fbk.eu

Bert Abrath,
Bart Coppens
Ghent University
Ghent, Belgium
{bert.abrath,bart.coppens}
@elis.ugent.be

ABSTRACT

Anti-tampering is a form of software protection conceived to detect and avoid the execution of tampered programs. Tamper detection assesses programs' integrity with load- or execution-time checks. Avoidance reacts to tampered programs by stopping or rendering them unusable. General purpose reactions (such as halting the execution) stand out like a lighthouse in the code and are quite easy to defeat by an attacker. More sophisticated reactions, which degrade the user experience or the quality of service, are less easy to locate and remove but are too tangled with the program's business logic, and are thus difficult to automate by a general purpose protection tool. In the present paper, we propose a novel approach to anti-tampering that (i) fully automatically applies to a target program, (ii) uses Remote Attestation for detection purposes and (iii) adopts a server-side reaction that is difficult to block by an attacker. By means of Client/Server Code Splitting, a crucial part of the program is removed from the client and executed on a remote trusted server in sync with the client. If a client program provides evidences of its integrity, the part moved to the server is executed. Otherwise, a server-side reaction logic may (temporarily or definitely) decide to stop serving it. Therefore, a tampered client application can not continue its execution. We assessed our automatic protection tool on a case study Android application. Experimental results show that all the original and tampered executions are correctly detected, reactions are promptly applied, and execution overhead is on an acceptable level.

Keywords

Software security; software attestation; remote attestation; code splitting; anti-tampering; tamper detection; tamper reaction

1. INTRODUCTION

Man-At-The-End attacks (MATE) have recently been defined to describe the cases where attackers tamper with software applications in contexts where they have full privileges (their own desktop computers or mobile devices) and they can deploy advanced tools to perform powerful and sophisticated analyses and changes.

In this paper, we cope with the problem of how to mitigate attacks aiming at modifying programs to alter their behaviour, for instance attacks to code integrity. For example, a procedure that performs authentication could be altered to grant unauthorized users access to critical resources, such as to freely generate temporary passwords with One Time Password generation available as a banking mobile app.

Anti-tampering is among the techniques available in the literature to mitigate the impact of the attacks we want to counter. Anti-tampering is usually composed of two components: detection and reaction. The tamper detection component usually consists of an attestation manager, in charge of collecting evidence that the application is working as expected, and of a verifier that emits verdicts on the program integrity based on this evidence. The tamper reaction component, instead, aims at punishing a detected tampering, e.g., turning the program unusable. Tamper detection may run locally, such as in *code guards* [9], where the attestation manager and the verifier are parts of the protected program. Alternatively, tamper detection may run remotely, like in the case of *remote attestation* [11], where the attestation manager is part of the protected application, but it sends proofs to a remote verifier that is executed on a trusted server.

Similarly, tamper reaction may work locally or remotely. When tamper reaction is local, it consists of pieces of code that worsen the performance or block the program, like *graceful degradation* [42]. Remote/server-side reactions consist in inhibiting the server from providing services to tampered clients. Local reactions are among the weakest points of anti-tampering. Indeed, they could be identified and disabled by an attacker, so that anti-tampering would become useless. Indeed, local reactions adopt patterns that are recognisable with static and dynamic analysis. Server-side reactions represent a more trustworthy solution. In fact, server-side reactions cannot be disabled by definition, because the remote server is assumed to be beyond the MATE attacker's control. However, server-side reactions are effective only if the program's correct execution directly depends on the server, and the server is not easy to fake. If a program does not depend on the server, server-side reactions do not work and, to the best of our knowledge, no methods are available to automatically transform programs to apply this kind of reaction. They, in fact, require substantial manual intervention to be tuned on the specific business logic of program to protect. A high degree of automation is required to make anti-tampering cheap and effective, and thus spur its adoption.

In this paper, we propose a novel approach to anti-tampering that we call *Reactive Attestation*. Our approach consists of automati-

SPRO'16, October 28 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4576-7/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2995306.2995315

cally (i) transforming a program to protect, which may originally have run without interacting with a remote server, into a program that strictly depends on a remote server; (ii) adding tamper detection mechanisms to the program to protect, and (iii) adding support to tamper reactions that stops tampered clients from working by preventing the remote server from serving them.

Our approach has been implemented in a prototype tool that works fully automatically on C code. This prototype makes the program to protect server-dependent using Client/Server Code Splitting, uses Static Remote Attestation as tamper detection technique, and uses a server-side tamper reaction logic that stops to serve detected tampered programs, thanks to a server-side communication infrastructure between tamper detection and reaction. This prototype has been subject to empirical validation, showing very precise tampering detection and acceptable memory and execution time overhead, and limited network communication bandwidth.

The paper is composed as follows. Section 2 introduces the background knowledge on anti-tampering and Client/Server Code Splitting we have used to design and build our approach to Reactive Attestation. Section 3 first introduces the limitations of current practice then presents the design principles of the Reactive Attestation. Section 4 describes the details of the implemented prototype. Section 5 presents the research questions we formulated, the results, the analysis, and a discussion on the experimental validation conducted on the implemented prototype. Section 6 presents related work and compares it with our Reactive Attestation. Finally, Section 7 draws conclusions and sketches future improvements.

2. BACKGROUND

This section presents the background knowledge on Client/Server Code Splitting and on Remote Attestation required to understand the motivations of their integration as Reactive Attestation.

2.1 Anti-tampering

Remote Attestation is a tamper detection technique that relies on a trusted server to request attestation and to verify proofs produced by the program. A generic reference architecture of Remote Attestation has been proposed by Coker *et al.* [11] (see Figure 1). The integrity of a *target* protected program is evaluated by an *attestation manager*, displaced in the application itself. The attestation procedure is initiated by a remote third party, namely the *appraiser*, which is in charge of requesting integrity measurements (also named integrity evidence) from the attestation manager, whenever it deems this appropriate. Integrity measurements are computed by the attestation manager and sent back to an *attestation delegation proxy*, which evaluates the measurements and decrees on the integrity of the target. Moreover, a *self-protecting trust base* behaves as a root of trust and is used by the attestation manager to certify its integrity measurements.

Remote Attestation uses challenge-response mechanisms to avoid replay attacks. The appraiser sends *attestation requests* containing nonces and, optionally, other commands for the attestation manager, to be used when producing integrity proofs. The attestation manager computes integrity measurements to be used as attestation evidence using the nonce and following, if available, the commands sent by the appraiser. Optionally, the computation of attestation evidence includes data that may permit the server to authenticate the target application. The self-protecting trust base plays a major role to ensure validity of application authentication data. The proxy, depending on the Remote Attestation technique, independently computes (or pre-computes and stores) the attestation evidence to be compared with the received one, or assesses though ad hoc integrity models if the evidence received proves the correct execution. In most cases, the communication between client-side and server-side components is protected with standard secure communication protocols to avoid Man-in-the-Middle attacks.

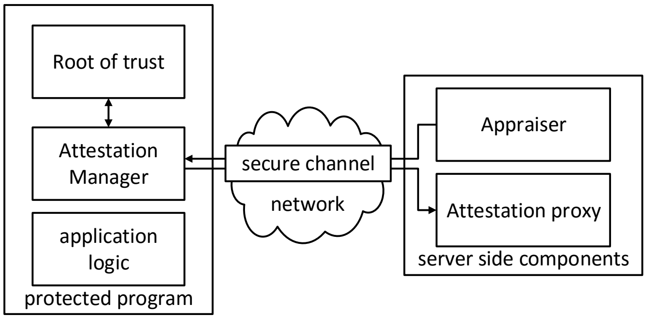

Figure 1: Remote Attestation architecture

2.2 Client/Server Code Splitting

Client/Server Code Splitting transforms a program such that part of the computation is moved to a remote server. This transformation was initially designed as a way to protect programs from malicious tampering [7, 8]. In fact, if sensitive portions or critical functionalities are moved to a remote server, they can run securely and without being tampered with by the attacker. The initial objective of Client/Server Code Splitting can be extended so that it can be used (possibly in combination with other ones) for new purposes.

This technique starts from the assumption that moving an entire function to the server is not always feasible. In fact, a function scope might not exactly match a security critical region. A function can be larger than the needed portion of code to be transferred, thus causing unnecessary server load if moved. Moving a whole function often has several side effects, such as updates to the program status (e.g., values of global variables), that would require intense client/server communication to propagate all the updates to the server. When computing what portion of the client to move to the server, data and control dependencies should be carefully taken into consideration to ensure minimal server overhead and performance impact, while guaranteeing an adequate level of client security.

To minimise the size of the code to move, Client/Server Code Splitting relies on the notion of *barrier slicing* [23], which extends the traditional concept of (backward) program slicing[1] [43].

The computation of the barrier slice starts with two sets of {program statement, program variable} pairs, namely the criterion $C = \{v_c, x_c\}$ and the barriers $B = \{v_b, x_b\}$. The algorithm starts by marking all the statements in the criterion x_c. Then control dependencies (on the statements x_c) and data dependencies (on the variables v_c at statements x_c) are traversed backward, and the reached statements are also marked. The algorithm is iterated, and dependencies are transitively traversed backward and new statements are marked. Dependencies, instead, are not traversed on barriers (statements x_b and variables v_b). When the propagation reaches the fix point, the computation of the barrier slice is over. The slice represents the part to be moved on the server.

When using Client/Server Code Splitting to protect code from malicious tampering, the identification of criteria and barriers should be based on security requirements. For example, the variables v_c in

[1] A backward slice s of a program P is a sub-program P' that is equivalent to the original program P with respect to the value of a particular variable v_c at a certain statement x_c (the pair $\{v_c, x_c\}$ is called the *slicing criterion*).

a criterion should be those whose computation must be protected. Barriers represent secure points where tampering is no longer an issue. They can pose a limit to the size of the slice. When Client/Server Code Splitting is used to make an application depend on a server, the scenario is much easier. In fact, it is sufficient to move to the server those parts that are complicated enough to avoid being faked by the attacker, and that are executed often enough to react promptly to tampering.

When the computation of the slice is available, a set of program transformations are applied:

- the slice is removed from the original application and moved to the server where it runs;
- each assignment to criterion variables v_c is replaced by a synchronization point, that both server and client should reach before the computation can proceed. Synchronization is implemented by exchanging particular *sync* messages between client and server;
- whenever the client requires the value of a variable whose computation is part of the slice moved to the server (e.g., v_c), this value is requested to the server via an *ask* message;
- when the server needs a value of a variable whose computation remains on the client because it is not part of the slice (e.g., v_b), this value is provided by the client via a *send* message.

3. REACTIVE ATTESTATION

In this section, we present our novel approach to code protection that we call Reactive Attestation. Reactive Attestation integrates the two techniques described in the previous section, namely Remote Attestation and Client/Server Code Splitting, with the objective of conceiving a brand new protection approach that overcomes the limitations in the reaction functionality of existing anti-tampering techniques.

Intuitively, Remote Attestation is applied to a program to protect in order to detect tampering, and a remote server component is deployed to request and verify integrity evidences. Then, Client/Server Code Splitting is used to transform the protected program and move selected parts of the (original) client to a secure server, so that the program to protect becomes server-dependent. The resulting protected program will be eventually composed of a client-side and a server-side logic.

Server-side components of Reactive Attestation, i.e., tamper detection and tamper reaction, are interconnected. Whenever a client is detected as tampered by Remote Attestation, the reaction is enforced by stopping the server-side components of Client/Server Code Splitting, i.e. the client is disconnected. This reaction blocks the normal execution of the protected program, turning it unusable.

3.1 Considerations on Remote Attestation

Remote Attestation is a broad category of protections that uses a remote server to move away from the client integrity verification and management functionality (see Section 2.1). Since only the attestation manager remains at risk, Remote Attestation is a rather secure protection. Moreover, since the protected program must send attestation evidences to the server, the attacker has just a limited amount of time to forge fake integrity evidences for the tampered program. However, Remote Attestation is not perfect. Attacks that aim at removing or disabling the protection can be mounted against the different Remote Attestation techniques. For instance, static techniques are vulnerable to memory copy attacks and they cannot detect attacks that do not alter binaries (like attaching a debugger). Time based attestation is vulnerable to proxy attacks and rely on trusted information about the precise client hardware configuration. Attestation based on dynamic techniques is subject to false positives and negatives (see Section 6.1 for the complete definition of Remote Attestation typologies). On the other hand, Remote Attestation is not very demanding from the point of view of server computation and network load. Additionally, it usually requires very little computational resources at the client side.

3.2 Considerations on Tamper Reaction

Tamper reaction may work locally, in which case it consists of pieces of code that worsen the performance of the application or block it (e.g., graceful degradation). Local reaction techniques are one of the weakest points of anti-tampering, as spotting and disabling the reactions renders the entire anti-tampering technique useless. Indeed, local reactions alter binaries or perform computations (mainly based on pointers) whose patterns are recognisable with static analysis and especially with dynamic analysis (see Section 6.2).

The only way to achieve a reasonable level of trustworthiness in anti-tampering is by using server-side reactions, where the punishment consists of preventing the server from providing services to compromised clients. Indeed, server-side reactions cannot be disabled, as the remote server is not under the control of MATE attackers.

Thus, server-side reactions are effective only if the client depends on the server and the code that runs on the server is cannot trivially be rebuilt locally by an attacker. Otherwise, stopping all the communications with the server or creating a fake server would be a valid way to defeat reactions. However, not all applications are designed to depend on a server, even though assuming that an application is always connected is more than reasonable nowadays. This is the main limitation of remote tamper reaction, and the reason for relying on Client/Server Code Splitting for obtaining a reliable reaction.

3.3 Considerations on Client/Server Code Splitting

Client/Server Code Splitting moves parts of a program on a remote server, where they are executed in parallel to the remaining parts of the client. The main limitation of Client/Server Code Splitting is related to performance. Indeed, it involves costs due to server-side computation and network load, which typically causes extra power consumption, especially in case the original unprotected barely used the network or did not access the network at all. Therefore, the need of minimising the size of the part to split becomes utterly important. On one hand, the portion of code subject to splitting should be small and simple enough to limit extra costs in server-side computation, and must not be executed too frequently, to limit extra costs in server-side computation and network load.

On the other hand, the split part should contain important features that are mandatory for the client, to prevent attackers from easily building a fake server that emulates the server-side logic. Split parts must be difficult to bypass (they should be scattered throughout the source code, and possibly have no relation to code parts annotated by Remote Attestation) and they must be executed often enough to be able to block compromised applications with a relatively short delay. Furthermore, split parts should not be code regions that return constant values, to avoid the possibility for an attacker to reuse those constant values in following executions and to remove the need of the computation done at server-side. Thus, it is not possible to generically split a large part of the program, but the part to protect should be identified carefully, typically by hand (which is costly and time consuming).

The identification of the most appropriate part to split can be automated, for example as suggested by Ceccato et al. [8]. They relied on static analysis to identify and minimise control and data dependencies that would be impacted by the transformation. Static analysis or the inspection of execution traces can be also used to set *criteria* and *barriers* near to the part of the program subject to remote attestation. In this way, the reaction to tampering would be quite near (in code and, so, in time) to the identification of tampering.

3.4 Requirement of Reactive Attestation

Based on the considerations elaborated so far, formulated on the strong and weak points of available protections and components, we set the requirements of an effective solution to Remote Attestation that will guide the definition of our novel code protection approach.

- *Automatic transformation.* The transformation of a program to adopt tamper detection should be fully automatic. Developers should just focus on developing and maintaining the program business logic and they should not spend effort on the protection.

- *Security requirements as code annotations.* The code areas to be protected from tampering must be explicitly annotated by the developers. The protection tool may suggest related areas, that is, other areas that deserve to be protected to achieve a better security level (e.g., to protect also the function calls, not only the function implementation). This requirement is in line with the state of the art in software protection, as currently there is no way to automatically determine the assets in an application.

- *Server dependent.* The program to be protected might not be network-oriented, so a remote tamper reaction would be unnatural and easy to spot. Such a program should be automatically transformed in semantically equivalent version for which remote reactions makes sense. More explicitly, this requirement translates into two goals: (i) turning the program dependent on the server so that stopping the service renders the application unusable; and (ii) making impossible for an attacker to generate a fake server which is able to mimic the correct behaviour of the server.

- *Accurate reaction.* When tampering is detected, the decision on reaction must be made automatically and fast. However, while all the tampered clients should be always blocked promptly, legitimate clients must not be disconnected, otherwise the protection would cause denial of service.

- *Acceptable overhead.* The overhead of the protected application (execution time, memory, network) should be limited, and it should not impact too much the user experience.

3.5 Reactive Attestation

Reactive Attestation consists in using Remote Attestation to check for tampering and Client/Server Code Splitting to implement the reaction. It consists in a static transformation of the program to protect and in some runtime logic to enact the actual protection.

3.5.1 Static Program Transformation

The process of transforming the program to protect is composed of the subsequent steps.

- *Security requirements as code annotation.* The developer specifies security requirements as the portions of code where integrity is needed. These parts are marked by means of code annotations added to the original program code (see Section 4.1 for the annotation syntax).

- *Automatic definition of split boundaries.* The boundaries of the code portion to split (i.e. the splitting configuration) are determined with an automatic process based on the system dependence graph (SDG) of the program that needs to be protected. In particular, control and data dependencies to be affected by splitting are minimised to limit the client/server interaction. Moreover, the split portion should be small in order to reduce the load at the server-side. The split configuration is marked as annotations in the source code to protect as *criteria* and *barriers* for the splitting algorithm (see Section 4.1 for annotation syntax).

- *Automatic splitting.* Client/Server Code Splitting is applied, to generate a transformed client-side component and a new server-side component, with proper *sync*, *ask* and *send* messages.

- *Automatic attestation.* The Remote Attestation transformation is applied, that means, the attestator manager is injected in the program to protect and all the server-side components for management and verification of integrity evidences are built. Client-side network management code is also injected into the program to protect.

- *Server-side orchestration logic.* All the necessary reaction logic needed on the server side is created. Server-side logic and server-side Remote Attestation components are updated to be aware of the presence of the reaction logic.

3.5.2 Runtime Protection

The interactions between tamper detection, tamper reaction and server-side logic follows this sequence.

1. *Verification.* The Remote Attestation verifier checks if the application is legitimate or not. All the verification results are recorded in a server-side data base.

2. *Reaction policy.* The reaction logic at the server side is based on an engine that analyses the content of verification history in the data base and decides whether to stop a specific client according to a reaction policy.

   A policy is indeed a set of rules that determine when the client program should be run and when it should not. For instance, a reaction policy may require to stop serving for x minutes an application whose last y proofs were invalid and to stop serving the application for $x + k \cdot t$ for all the successive failed t attestations.

   The reaction logic updates the tampering status of the application in the server-side data base.

3. *Reaction.* When the split server execution is required by a protected client, the split server checks the application's tampering status stored in the server data base. If the tampering status confirms that application is *"valid"*, the execution of the split server is granted and proceeds as usual. Otherwise,

```
1  void check_license(int day, int month, int year) {
2     _Pragma("RA begin attestation(technique(static_ra),
          static_attestator_type(a1),interval(180),
          attest_at_startup)")
3     int dd1 = calculate_original(day, month, year);
4     int dd2 = calculate_current()
5     _Pragma("RA end")
6     if (dd2 - dd1 > 30)
7        printf("Fail\n");
8     else
9        printf("Ok\n");
10 }
11 int calculate_original(int d, int m, int y) {
12    _Pragma("RA begin splitting(barrier(valid_year), label
          (s1))")
13    int valid_year = 0;
14    _Pragma("RA end")
15    valid_year = check_valid();
16    _Pragma("RA begin splitting(criterion(original_date),
          label(s1))")
17    int original_date = d + m + y + valid_year;
18    _Pragma("RA end")
19    return original_date;
20 }
```

Figure 2: Example of source code annotations for Reactive Attestation.

if the status is *"tampered"*, the split server code is not executed. Since split client and split server executions are synchronized with bidirectional data exchange, when the server-side execution stops, the client-side execution also blocks. The application then becomes unresponsive and is thus no longer usable.

4. PROTOTYPE IMPLEMENTATION

This section describes the implementation of a prototype tool to perform Reactive Attestation, as to validate the approach presented in Section 3. The implemented prototype includes Client/Server Code Splitting and Remote Attestation modules that work with source files written in C. Therefore the only limitation we imposed, compared to the requirements in Section 3, is that the application to protect includes at least a part to protect which is written in C. The other limitation is that the application must be executed only when the device on which it runs is connected to the Internet in order to reach the server-side components. All the applications that satisfy these requirement can be protected with the Reactive Attestation.

4.1 Code Annotations

Source code annotations drive the program transformation. Figure 2 shows the syntax of annotations used by Reactive Attestation. Annotations are defined by means of the *pragma* construct, they are directive to the C compiler, starting with the string *"RA"*. Whenever the annotated code is compiled by the standard C compiler, which does not define semantics for *"RA"* annotations, a warning is emitted but the code is correctly compiled without applying any protection, generating the original, vanilla version of the application. When the program is compiled by our tool, however, the resulting binary is protected by Reactive Attestation.

Annotations apply to the portions of code delimited between *"RA begin"* and *"RA end"*. Two annotation variants are available, to control parts subject to Remote Attestation (i.e., *"RA begin attestation"*) and parts affected by Client/Server Code Splitting (i.e., *"RA begin splitting"*).

```
procedure calculate-slice (criterion C, barrier B)
  set-of-vertices := vertices-of (C)
  set-of-barriers := vertices-of (B)
  slice := vertices-of(C)
  while not (is-empty(set-of-vertices))
        predecessors := predecessors-of-vertices (set-of-
            vertices, DATA-CONTROL-DEPS)
        filtered-predecessors := predecessors \ set-of-
            barriers
        set-of-vertices := filtered-predecessors
        slice := slice UNION filtered-predecessors
```

Figure 3: Pseudo-code of the barrier slicing algorithm.

The attestation annotation in Figure 2 indicates that the code from line 2 to 5 will be attested for modifications with Static Remote Attestation. In particular, we have requested the use of a specific attestator (we have 40 variants developed within the ASPIRE project[2]) through the optional `static_attestator_type` keyword. This code area is subject to attestation with an average frequency of one attestation every 180 seconds through the `interval` keyword. Moreover, this area is attested when the program is launched via the `attest_at_startup` keyword.

Splitting annotations, instead, specify a slicing criterion (lines 16 - 18) and a slicing barrier (lines 12 - 14). Barrier and criterion annotations also indicate the variables subject of the splitting algorithm by name, they are respectively `valid_year` as barrier and `original_date` as criterion. Moreover, splitting annotations use the *"label"* parameter to pair barrier and criterion annotations, which is mandatory when splitting is applied multiple times to the same program.

4.2 Client/Server Code Splitting Module

The Client/Server Code Splitting module is implemented in a number of steps. It mainly combines Grammatech CodeSurfer[3] and the TXL transformation framework to analyse the source code of the application that needs to be transformed [13], to identify those portions of the code that require to be moved onto the secure server, and to apply code transformation patterns to generate the transformed client application.

The implementation of the proof-of-concept prototype described in this paper misses the automatic decision of the optimal slice extent. Although automation can be implemented according to algorithm available in the literature [8], the current implementation requires to manually annotate slicing barriers and criteria. Once barriers and criteria are available, the computation of the slice and all the program transformation steps are fully automated.

Input of the tool is the annotated source code, while the output consists of the sliced client and the corresponding server-side slice component.

The component responsible for the computation of the barrier slice is implemented as an analysis script on top of CodeSurfer. Starting from the information extracted from annotations, the slicing criterion and the barriers, we implemented the slicing algorithm to precisely calculate the portion of code that represents the barrier slice with respect of the current annotation configuration and the code to protect. The code in input is analysed by CodeSurfer to extract the system dependence graph (SDG). The slicing algorithm performs a series of queries to this data structure to extract the barrier slice.

Figure 3 shows the pseudo-code of barrier slicing. Input parameters are the slicing criterion C and the barrier B. These parameters are converted into their representations in terms of vertices of the SDG and stored in two local variables, *set-of-vertices* and *set-*

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

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

of-barriers respectively (line 2, 3). At this point, the barrier slice (variable *slice*) is initialized to the vertices that represent the criterion C (line 4). Then, the algorithm iterates until no new vertex can be added to the slice (i.e., the fix point is reached). More specifically, in the first iteration the algorithm queries all the predecessors of the current set of vertices (representing the criterion C) by following data and control dependencies backward in the SDG. The set of predecessors is filtered by removing possible barriers. When a barrier is found, the propagation of dependencies stops and vertices at that barrier are not included in the slice. After filtering, the resulting set of vertices becomes the set of vertices for the next iteration. When no new predecessor is found, the algorithm stops and returns the final barrier slice.

When the slice is computed, a set of program transformation rules is applied, which have been implemented on top of TXL as rewriting transformations of the parse tree of the program. These transformation rules are used in order to:

- create a new compilation unit for the new component that needs to be deployed on the split server;
- transform the client program such that it only works when paired with the corresponding split server.

4.3 Remote Attestation Module

The reference architecture of our software only Static Remote Attestation system is composed of three main components, based on the work of Coker *et al.* (as in Figure 1) and on the analysis of the literature presented in Section 6.1.

The *Manager* is the appraiser in charge of initiating the integrity evaluation procedures. It prepares and sends requests to the attestator. Requests contain indications on how to perform the attestation and a nonce used for countering replay attacks. The frequency of requests is stochastically described, i.e., the time between two attestation requests is defined by an average expected value and a variance. Namely, the variance is statically defined as 2% of the average expected value.

The *Attestator* is the attestation manager that computes the integrity measurements on the target. The integrity measurements, that are the integrity proofs, are the hash of data obtained as random walks into selected portion of the target code memory and nonces sent by the Manager. As a special case, the random walk can be performed on the entire application. The actual portion of the code memory to attest, the type of random walk and the hash algorithm to use are requested by the Manager. The obtained measurement is then hashed and sent to the Verifier.

The *Verifier* is the attestation proxy. This element compares the integrity evidences received from the attestator against the precomputed valid values and emits a verdict about the target integrity.

The *Reactive Attestation data base* logs all the attestation transactions that include sent requests, evidences as well as the verdicts, so that the complete history of all the evaluations is tracked. This information is accessed by the policy engine to determine if a reaction is needed and its severity.

All the messages exchanged by the components over the network are protected by using secure communication channels.

The attestation evidence is computed as:

$$e = \mathrm{hash}(\mathrm{prepare}(d, \mathrm{ID}, \mathrm{n}))$$

where ID is a valid target identification data, and n is the nonce received from the server. The integrity measurement data d are computed as a random walk on a memory code area ($d = \mathrm{random\_walk}(a)$). The random walk function to use is determined by the nonce (i.e., $\mathrm{random\_walk} = f_r(n)$). The data to hash are concatenated and manipulated by the $\mathrm{prepare}$ e.g., to compute a keyed digest n that is put at the begin and the end of the data to hash.

Information about the code areas to attest is stored in an Attestation Areas Data Structure (ADS). The code areas to attest are not in general contiguous segments of the code memory and are thus represented as sequences of contiguous memory blocks. Information about each memory block is determined as an offset from the application base address (also named “load address of the text segment” in the ARM specifications[4]). Areas to attest are associated with unique integer identifiers. The actual area to attest is derived from the nonce by means of an ad hoc function $a = f_a(n)$. An API provides access to the i-th byte of the j-th code area. Therefore, the ADS can be injected into the application binary code either as a contiguous blob or spread across the binaries. The injection is actually performed by means of the binary rewriting features of Diablo[5], which is a link-time rewriting framework.

4.4 Reaction Policy Engine and Reactive Attestation Data Base

The Reactive Attestation data base has been implemented as a standard relational data base. It includes the following tables:

- Area, which reports the ID of all the areas, the attestation average frequency, and if the area needs to be attested at startup;
- Request, which reports about the area ID, request time, response time, verification results, and all other parameters;
- Reaction_status, which reports the allowed values for the overall application status, depending on the reaction policy;
- Reaction, which reports the overall status of clients as established by the reaction policy engine;
- Prepared_data, which stores the association between nonces, code areas, and pre-computed attestation data that serve to save time at verification time;
- Policy, which stores the association between an application and the reaction policy to enforce.

The reaction engine has been trivially implemented as a set of processes, one for each used policy, which accesses the data base and searches information about the programs that need to react with the algorithm implied by the policy. Clearly, this approach is acceptable for a prototype and it is not expected to scale to real systems. However, it can be easily improved with an event-based architecture. Every policy process accesses the request table and write the results (one of the reaction_status values) in the reaction table. We have implemented three sample policies: “stop serving a tampered application”, “stop serving a tampered application until it is restarted”, and the “stop serving for x minutes an application whose last y proofs were invalid and to stop serving the application for $x + k \cdot t$ for all the successive failed t attestations”.

5. EXPERIMENTAL VALIDATION

5.1 Research Questions

This section is meant to investigate Reactive Attestation from an empirical point of view, according to the following research questions.

[4] http://infocenter.arm.com/help/topic/com.arm.doc.dui0101a/DUI0101A_Elf.pdf

[5] http://diablo.elis.ugent.be/

- **RQ**1: is Reactive Attestation effective in detecting tampering?
- **RQ**2: what is the overhead of Reactive Attestation?

The first research question is devoted to explore the *accuracy* of Reactive Attestation in detecting real cases of code tampering. In fact, the protection should block only the programs that are under attack, while all the legitimate and original programs should not be impacted. The second research question investigates the *cost* of the protection, in terms of execution overhead of the protected clients with respect to the original clients.

It would also be interesting to pose questions on how long a tampered program can be executed before it is detected by the attestation and on the delay between detection and reaction. However, to correctly address these other questions, we would need a more advanced experimental environment, with different case study applications. A more detailed and in depth experimentation goes beyond the scope of our preliminary assessment, so other research questions are left as future work.

5.2 Metrics

To measure the effectiveness of Reactive Attestation we collect the following metrics:

- *True Positives (TP),* the number of tampered clients that are correctly blocked;
- *False Positives (FP),* the number of legitimate clients that are incorrectly blocked;
- *True Negatives (TN),* the number of legitimate clients that are correctly executed as legitimate;
- *False Negatives (FN),* the number of tampered clients that are incorrectly executed as legitimate.

These metrics are key metrics in information retrieval as they measure the performance of a classifier to correctly classify documents, so they are useful for us to understand the ability of our tool to distinguish between legitimate and unauthorised application executions. In our context, we want to maximise the ability of our Reactive Attestation tool in detecting True Positives (TP), those applications that are real cases of tampering in order to block their executions, and True Negatives (TN), those applications that are legitimate and which executions must be allowed. While maximising TP and TN, we also want to minimise the number of False Positives (FP). A FP means that a legitimate application has been blocked, negating an authorised user his/her rights to access a service. Furthermore, the number of False Negatives (FN) must be minimised too in order to avoid malicious users accessing restricted contents.

To assess the overhead caused by applying Reactive Attestation, we measure and collect the following metrics.

- *Memory (MEM)* is the amount of memory required to execute the (original and protected) program. Memory is measured by the `time` utility embedded in our experimental device (see following Section for details about the device used for the experiment). The `time` command runs a program, and displays information about the resources, like memory and time, consumed by that program.
- *Execution time (TIME)* is the time required to run the program. As done with MEM, TIME is also measured by means of the `time` system utility.
- *Network usage (NET)* is the amount of data exchanged by the program in a complete run. NET is directly measured at server-side by the Remote Attestation and Client/Server Code Splitting components.

The goal of applying Reactive Attestation to a program is to reduce the attack surface that can potentially be targeted by attackers. This, however, introduces modifications in the protected application. These changes have an impact on the overall performances of the application, in particular on execution time and memory occupation: with the metrics listed earlier, we aim to estimate the magnitude of that impact.

5.3 Experimental Procedure

The experimental assessment has been conducted on a case study program, namely *License*. License is an Android app written in Java, with a security critical part written in C. This critical part is a routine devoted to check the validity of a license number to activate a software component. Reactive Attestation will be deployed on the C part to protect it against potential tampering attacks. The native code is composed of two distinct functions, for a total of 105 lines of code.

Considering the security requirements for this case study, we manually define code annotations to specify where Remote Attestation should check for tampering and what is the mandatory feature that should be subject to Client/Server Code Splitting for implementing the reaction. In particular, Remote Attestation checks the function that actually verifies the validity of the license, while Client/Server Code Splitting transforms an utility decoding function used by the check, such that, if the secure split server is disconnected, the client does not work. With the selected annotation configuration, each line of code is transformed by at least one of the two steps, Remote Attestation or Client/Server Code Splitting. This means that the two protections combined protect the 100% of the security critical code.

We created seven different tampered versions of the License app. We obtained these by mutating the C binary part of the protected app. In particular, to simulate a real attacks, we use a binary rewriting tool to alter the code that checks the license validity. The mutations include attacks that skip the check (overwrite with NOPs), that force a date as current date in the license period, or that alter the licence expiration date.

We then execute the original and each of the tampered programs produced by our tampering tool. Each execution is performed on a NITROGEN6X development board, equipped with an i.MX6 ARM-Cortex A9 processor, 1 GHz clock speed, with 1 GB of 64-bit wide DDR3 at 532 MHz and with Android 4.3 Jelly Bean installed. Server-side components run on a virtual machine with 4 processor cores, 4 GB RAM, with Debian 7.7 installed. We trace if the program is executed correctly or if its execution is stopped by the server in case of tampering detection. Moreover, for the vanilla (the original, unprotected) application and for the protected, untampered version, we collect the overhead in terms of memory, execution time and network usage. To have a reliable measurement of time, we wrapped the original C code into a loop, to execute the native part for 100 times (which means that the license check is performed 100 times before exiting). We defined two distinct usage scenario, one in case of valid license and another one in case of invalid license. Each scenario is then executed 15 times, for a total of 30 times per application version.

5.4 RQ1: Reactive Attestation Effectiveness

Table 1 shows the effectiveness of Reactive Attestation in terms of True/False Positives and True/False Negatives. As we can see, Reactive Attestation correctly reports the protected, non tampered version of the application as True Negative (column 4). When executed, the protected app is checked but its execution is allowed because no malicious tampering is detected. All the 7 tampered versions are correctly classified as True Positives (column 2).

Table 1: Effectiveness of Reactive Attestations

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

Thus, according to our experimental settings, Reactive Attestation managed to correctly identify and grant execution to all the legitimate clients. It also managed to correctly identify and block all the tampered clients.

5.5 RQ2: Reactive Attestation Overhead

The measured overhead of Reactive Attestation is shown in Table 2. For the original and for the protected programs, and for both usage scenarios (column 1), the Table reports the *mean* and *standard deviation* of absolute values of Memory (MEM, column 2), execution time (TIME, column 3) and network usage (NET, column 4).

For both versions of License, original and protected, we added an artificial loop of 100 executions of the native code. This is done for two reasons, a) to measure the execution time of the vanilla version of the app, which otherwise executes too fast to be measured accurately, and b) to avoid that constant setup time required to start a process dominates the actual execution time. With this modification, execution time can be measured in a more precise way.

As can be seen in Table 2, memory usage for the original, vanilla version of the program is 2192 kB in case of checking a valid license, and 2176 kB otherwise. In case of the protected app, the memory usage increases to 13 715 kB for the valid license scenario, and 13 960 kB for the invalid one. The increment in memory usage we registered is roughly 11 000 kB, and it is caused by the setup of the network communication infrastructure used by Reactive Attestation to make client-side and server-side communicate, and by the insertion of the attestation manager and its Attestation Area Data Structure. The size of all these components, except for the ADS, are constant and do not depend on the application or the protection configuration chosen. The size of the ADS will scale linearly with the amount of binary code regions that can be attested.

Execution time for the original version in both the usage scenarios is 17 ms and 18 ms. This means that the execution time for a single run of the vanilla app can be roughly calculated by dividing the obtained times by 100, which gives a result of 170 µs for the valid license scenario, and 180 µs for the invalid license scenario. For the protected version, the execution time for the two scenarios is 8.441 s and 8.506 s. By dividing this time by 100, we obtain an execution time for single run which is 84 ms and 85 ms.

Network load is constant across the usage scenarios, for a net payload of 392 B. This represents the exact amount of net data the Reactive Attestation protection needs to exchange from client to server (and vice versa when required) for a single run of the application, without considering data used by the communication protocol to initialise and establish the connection between the two ends.

5.6 Observations

Based on the quantitative results collected in the experimental validation, we can formulate the following qualitative observations. We highlight here that it is not possible to estimate the impact of this protection on a generic application without having execution traces. Indeed, the performance overhead depends on how often the split points are encountered and on the complexity of the split code. Nevertheless, to avoid these performance issues, we propose to choose the parts to split by analysing the execution traces in order to maintain the overhead to an acceptable level.

Reactive Attestation is effective.

Reactive Attestation blocks all the executions of malicious copies of the application under analysis. At the same time, the approach did not block executions of the legitimate version of the application. This result suggests that Reactive Attestation is effective in detecting tampered applications, without affecting the experience of legitimate users. However, it is important to note that the use case subject of the experiment is small and probably not fully representative of real-world applications. Moreover, Reactive Attestation is vulnerable to all the attacks that defeat the Remote Attestation, that is, attacks that allow forging valid evidences from invalid applications. The Static Remote Attestation is known to be relatively fragile but the Reactive Attestation is independent on the actual tamper detection technique used.

Reactive Attestation requires time.

We observed a huge difference in the amount of time required by the vanilla application and the protected application to complete. While the vanilla program is very fast, the protected program took a lot more, even if execution time for the protected program in case of a single run is also very small and the delay caused by the protection is barely perceivable by the user. It should be noted that we chose to protect the entire application (100% of code protected) instead of focusing on specific variables or assets. In fact, if we consider the case of a program developer, she/he might want to protect peculiar, small parts of her/his application. Focusing only on small portions of the application would drastically reduce the impact of Reactive Attestation on the execution time. However, in case of our License application 100% of protected code means 105 lines, a quantity which is roughly comparable to the dimension of security critical areas and assets in real-world applications. In case of bigger applications, the overhead caused by Reactive Attestation on the execution time would be most probably dominated by the time required for the regular execution, which is usually much larger than few microseconds.

Reactive Attestation has small network usage.

The protected application and the server-side components exchanges 392 B per single execution. This indicates a small network consumption, and suggests that Reactive Attestation could have a limited impact on user's data plans to be consistently use as protection for mobile applications. The amount of exchanged data, however, strongly depends on the functionalities and what code a developer might need to protect. Using Client/Server Code Splitting to protect a functionality that is computationally intensive and that handles lots of data can highly increase the network occupation of the application. Protection configurations must be carefully chosen to reduce the impact on the network.

Table 2: Overhead of Reactive Attestations

Variant (scenario)	MEM		TIME		NET	
	Mean (kB)	SD	Mean (s)	SD	Mean (B)	SD
Original (valid license)	2,192	0	0.017	0.006	-	-
Original (invalid license)	2,176	0	0.018	0.004	-	-
Protected (valid license)	13,715	145.540	8.441	0.290	392	0
Protected (invalid license)	13,960	33.598	8.506	0.323	392	0

6. RELATED WORKS

6.1 Remote Attestation

At a very high level, work about remote attestation can be classified as solutions that rely on hardware components as a self-protecting trust base and software-only solutions. One of the first and most common applications of the remote attestation is the *Integrity Measurement Architecture (IMA)* proposed by the Trusted Computing Group (TCG) [12, 34]. The TCG realisation exploits a hardware module, the *Trusted Platform Module (TPM)*, which acts as the self-protecting trust base. As the technology evolved, the TCG approach evolved as well, thus TCG has extended the hardware-based solution to virtual and cloud-based environment [30, 35]. Other authors proposed a secure processor architecture based on Phisically Unclonable Functions [41]. On the commercial side, Intel proposed the Trusted eXecution Technology (www.intel.com). Another work from Basile *et al.* [5] proposed the use reconfigurable devices (FPGA) as a HW self-protecting trust base. In their approach, Attestation Managers are FPGA cores able to attest the executed binaries in memory by directly accessing them without being mediated by the operating system. All these hardware-based solutions rely on the fact that programs to protect already depend on a remote service. They could take advantage of our Client/Server Code Splitting approach in order to be applied to more scenarios. In general, hardware-based solutions could be used as tamper detection with reactive attestation, even if, given their requirements, they barely apply to mobile and embedded scenarios.

Beside the HW-based solutions, software-only mechanisms have been proposed that are better suited for mobile and embedded systems. Armknecht *et al.* [3] have coined the definition of software attestation, to distinguish software-only approaches from the ones that follow the TCG approach. Software-only approaches are divided in three categories, depending on the properties used to compute and verify attestations: time-based attestation, where being able to compute an answer in time is indeed the integrity evidence; static attestation, where static properties of the application are considered, like binaries and read-only memory properties; and embryonal works that use dynamic properties to infer execution correctness.

Time-based approaches estimate a time limit within which the evidence must be produced and sent to the verifier, if the time exceeds this estimation the evidence is not accepted. Seshadri *et al.* [38] realized their prototype, named Pioneer, based on time-based attestation. Integrity, with their solution, is assessed through the precise estimation of the execution time of precise code fragments executed as attestation proofs. Therefore, the need for hardware has been replaced with the need for a more precise relation between the software to execute and its execution environment, to avoid proxy attacks [26]. To the best of our knowledge, the binding to the execution platform cannot be effectively proved without the use of hardware to ensure OS Kernel integrity (like COPILOT [31]), which indeed becomes the new self-protecting trust base.

To our knowledge, the earliest proposal of static attestation is the Spinellis' software reflection, which proposed the hash of random parts of the memory [40]. Similar solutions are SWATT, proposed by Seshadri *et al.*, a software-based Remote Attestation that monitors target code memory regions [39], and MobileGuards, proposed by Grimen *et al.*, short-lived attestation agents downloaded from a trusted server [17]. Kennel *et al.* proposed a set of genuinity tests based on static information [20]. Indisputable Code Execution (ICE) is used in [36, 37] to build a root of trust and an untampered execution environment. Despite ICE implementation has been proven vulnerable [6], ICE has inspired Conqueror, another software-based scheme where the attestation manager receives an encrypted and obfuscated routine to compute the evidence [27], and SBAP, designed for resource limited peripherals like keyboards but it is likely suitable also for sensors [25]. Armknecht *et al.* have proposed to use a static approach that also limits the acceptance of the provided evidence on the basis of response time [3]. All the static remote attestation techniques can be considered as alternative techniques of the Static Remote Attestation we used for tamper detection. However, no one of the analysed papers focused very much on effectiveness of the reactions. Moreover, to the best of our knowledge, no one of the referenced papers focused on automatic application of Remote Attestation on generic applications, which is one of the main achievements of this work.

The Remote Attestation techniques presented so far aim at attesting static properties of the target, i.e., they only check target components' execution independent properties, like binaries images, configurations and read-only memory related properties. Sadeghi *et al.* proposed to focus on software properties instead of proposing new attestation mechanisms [32, 10]. This approach opened up the way towards dynamic attestation and has been applied to build a virtual TPM and a property-based bootstrap architecture [33, 22]. Similarly, Li and Shen [24] proposed model-based attestation as an alternative technique to static and property-based attestation, approach that was later generalized by Alam *et al.* [2] in what they call *Model-Based Behavioral Attestation (MBA)*. Both works are based on the UCON model [29]. The goal of MBA is to provide a method to attest that a remote platform will comply with a particular usage model when handling some objects. Gu *et al.* [18] measure target integrity by tracking the system calls invoked by the target during its execution and comparing the usage profiles with precomputed valid values obtained by a preliminary static and dynamic analysis. Furthermore, Abadi proposed to attest software based on CFG information [1]. Dynamic remote attestation has been also implemented by exploiting software *invariants*. Kil *et al.* [21] proposed a remote attestation system that monitors targets by checking data integrity throughout *data likely invariants* evaluation. They automatically extract data invariants by using a modified version of Daikon [15] that empirically deduces data invariants by analysing execution traces of the target. Baliga *et al.* [4] proposed a remote attestation system that detects rootkits inside an operating system by checking the validity of pre-computed data invariants on the data structures values with pre-computed data invariants (extracted with Daikon). Dynamic techniques are certainly promising but currently they suffer from false positives/negatives, as in several cases it is not possible to infer with reasonable precision that an attack actually alters some of the dynamic monitored properties. Nevertheless, in the future, they can be selected as alternative techniques in our framework thus taking advantage of the tamper reaction framework.

6.2 Tamper Reaction

The field of tamper reaction has not been widely explored and the proposed solutions do not have a high level of maturity. Trivially, a reaction mechanism may be to stop the software execution as soon as tampering is detected, thus collapsing detection and reaction in the same place. Unfortunately, this kind of reaction is not very effective because the effect – a software crash – can be easily associated to the cause – the tampering. Thus, this makes it easy to pinpoint the reaction part of the application and consequently to disable it. Indeed, Tan *et al.* [42] defined a basic principle for tamper reaction and its relations with tamper detection: tamper detection and tamper reaction must be separated in space and time. Code for detection and reaction must be placed in different parts of the application and must be hard to tell from rest of application code. Moreover, they must be executed at different moments, to decouple them and to avoid that a cause-reaction relationship can be inferred. In the same work, the authors proposed a delayed injection of software failures to "gracefully degrade" the normal execution of a tampered application. Oishi *et al.* [28] also proposed an execution degradation approach: they modify the original application binary by replacing (camouflaging) a set of instructions that get restored (de-camouflaged) at runtime only if the integrity of the application is still valid. If evidence of tampering is detected, the system does not restore the original instructions correctly, thus making the program run differently than expected, which includes buggy behaviour and errors. A self-correcting system has been proposed by Jakubowski *et al.* as an alternative to degradation or interruption [19]. The proposed reaction mechanism invalidates the effect of tampering by correcting it and replacing the changes with the original code by means of redundant code blocks and a voting system.

All the proposed reaction mechanisms apply changes to the running code in order to worsen or correct its behaviour. Given that the changes made by the reaction inevitably reside on the attacker's (untrusted) environment, they could be spotted – sooner or later. In our proposal, we move the reaction away from the untrusted environment, thus impeding any form of counter reaction. Remote reactions that stop serving compromised applications has certainly consistently used in practice (e.g., on demand media streaming services). Moreover, we are not aware, to the best of our knowledge, of tools that can automatically apply remote tamper reaction techniques to selected applications.

6.3 Client/Server Code Splitting

Our Client/Server Code Splitting approach is close to work proposed by Zhang and Gupta [44] to protect software against software piracy. In their work, a standalone application is turned into a network application by stripping fragments of code from *open*, insecure components to *hidden*, secure components. Fragments are selected via slicing to maximise the complexity for an attacker to reconstruct the original application. Differently from them, we adopted the concept of barrier slicing in order to reduce the size of code fragments that are moved to the secure server. This helps in limiting the workload at server-side and also to minimise network traffic generated by the protected application.

The concept of barrier slicing was also used by Ceccato *et al.* [7, 8] to design a protection technique against malicious tampering, with the idea of using barrier slicing to ensure that security critical portions of client computation are executed on the secure server. In our work, barrier slicing is applied to required although less critical functionalities, to turn the application server dependent and to perform the reaction is case of tampering detection.

Dvir *et al.* [14], instead, proposed to split applications into two new programs, a client that performs the application's active tasks, and a server that carries out lazy tasks. They identified memory allocation as lazy task to split, to make application and server communicate asynchronously. Our approach generates a client and a server that communicate in synchronous way, so to immediately react in case of tampering.

Another code splitting approach was proposed by Fukushima [16]. A separation technique is applied to the program to protect, to divide it into two pieces, a user program and a protected program, in order to protect applications that run in untrusted cloud computing environments. They defined a set of code transformations to encode/decode variables that need to be secured on the trusted server. Our tool, instead, uses an algorithm that is based on barrier slicing to separate the client from the server.

To the best of our knowledge, we do not know any tool that automatically applies code splitting to C code.

7. CONCLUSIONS

In this paper we presented Reactive Attestation, a novel approach for software protection that integrates Remote Attestation for tampering detection and Client/Server Code Splitting for tampering reaction. Once the security requirements are added to the code in form of annotations, our prototype tool automatically outputs a protected client and the corresponding server-side components. The program to protect is turned server-dependent by moving part of its execution to a remote server. Our approach reacts to tampering by denying server-side execution to clients that do not pass attestation checks. Experimental validation shown that Reactive Attestation is capable of accurate tampering detection with acceptable performance overhead.

As future work, we intend to extend the assessment of Reactive Attestation on real-world programs, and to study the server load when more and more clients connect at the same time. Moreover, we intend to study if, and to what extent, Reactive Attestation can be detected and defeated by automatic tools and by expert industrial hackers.

Acknowledgement

This research has been funded by the European Union 7th Framework Programme (FP7/2007-2013), under grant agreement number 609734 - ASPIRE project (Advanced Software Protection: Integration Research and Exploitation), https://www.aspire-fp7.eu/.

8. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):1–40, 2009.

[2] M. Alam, X. Zhang, M. Nauman, T. Ali, and J.-P. Seifert. Model-based behavioral attestation. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, SACMAT '08, pages 175–184, New York, NY, USA, 2008. ACM.

[3] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. A security framework for the analysis and design of software attestation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1–12. ACM, 2013.

[4] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 77–86. IEEE, 2008.

[5] C. Basile, S. Di Carlo, and A. Scionti. FPGA-based remote-code integrity verification of programs in distributed embedded systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(2):187–200, 2012.

[6] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 400–409, New York, NY, USA, 2009. ACM.

[7] M. Ceccato, M. Dalla Preda, J. Nagra, C. Collberg, and P. Tonella. Barrier slicing for remote software trusting. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 27–36. IEEE, 2007.

[8] M. Ceccato, M. Dalla Preda, J. Nagra, C. Collberg, and P. Tonella. Trading-off security and performance in barrier slicing for remote software entrusting. *Automated Software Engineering*, 16(2):235–261, 2009.

[9] H. Chang and M. Atallah. Protectioning Software Code by Guards. In *Proc. ACM Workshop Security and Privacy in Digital Rights Management, ACM Press*, pages 160–175, 2001.

[10] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A.-R. Sadeghi, and C. Stüble. A protocol for property-based attestation. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing*, STC '06, pages 7–16, New York, NY, USA, 2006. ACM.

[11] 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*, 10(2):63–81, 2011.

[12] T. Committee et al. Trusted computing platform alliance (tcpa) main specification v1. Technical report, 1b. Technical report, TCPA Alliance, 2002.

[13] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow. Txl: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[14] O. Dvir, M. Herlihy, and N. N. Shavit. Virtual leashing: Creating a computational foundation for software protection. *J. Parallel Distrib. Comput.*, 66(9):1233–1240, Sept. 2006.

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

[16] K. Fukushima, S. Kiyomoto, and Y. Miyake. Towards secure cloud computing architecture - a solution based on software protection mechanism, 2011.

[17] G. Grimen, C. Mönch, and R. Midtstraum. Tamper protection of online clients through random checksum algorithms. In *Information Systems Technology and its Applications, 5th International Conference ISTA'2006, May 30-31, 2006, Klagenfurt, Austria*, pages 67–79, 2006.

[18] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei. Remote attestation on program execution. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 11–20. ACM, 2008.

[19] M. H. Jakubowski, C. W. N. Saw, and R. Venkatesan. Tamper-tolerant software: Modeling and implementation. In *International Workshop on Security*, pages 125–139. Springer, 2009.

[20] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 21–21, Berkeley, CA, USA, 2003. USENIX Association.

[21] 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 *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 115–124. IEEE, 2009.

[22] R. Korthaus, A.-R. Sadeghi, C. Stüble, and J. Zhan. A practical property-based bootstrap architecture. In *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*, STC '09, pages 29–38, New York, NY, USA, 2009. ACM.

[23] J. Krinke. Barrier slicing and chopping. In *SCAM*, pages 81–87, 2003.

[24] X.-Y. Li, C.-X. Shen, and X.-D. Zuo. An efficient attestation for trustworthiness of computing platform. In *Proceedings of the 2006 International Conference on Intelligent Information Hiding and Multimedia*, IIH-MSP '06, pages 625–630, Washington, DC, USA, 2006. IEEE Computer Society.

[25] Y. Li, J. M. McCune, and A. Perrig. SBAP: Software-Based Attestation for Peripherals. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (Trust 2010)*, June 2010.

[26] Y. Li, J. M. McCune, and A. Perrig. Viper: Verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 3–16, New York, NY, USA, 2011. ACM.

[27] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: Tamper-proof code execution on legacy systems. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'10, pages 21–40, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] K. Oishi and T. Matsumoto. Self destructive tamper response for software protection. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 490–496. ACM, 2011.

[29] J. Park and R. Sandhu. Towards usage control models: Beyond traditional access control. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, SACMAT '02, pages 57–64, New York, NY, USA, 2002. ACM.

[30] R. Perez, R. Sailer, L. van Doorn, et al. vtpm: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium*, pages 305–320, 2006.

[31] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. COPILOT - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th conference on USENIX Security Symposium*, pages 13–13, 2004.

[32] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *Proceedings of the 2004 Workshop on New Security Paradigms*, NSPW '04, pages 67–77, New York, NY, USA, 2004. ACM.

[33] A.-R. Sadeghi, C. Stüble, and M. Winandy. Property-based tpm virtualization. In T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee, editors, *Information Security: 11th International Conference, ISC 2008, Taipei, Taiwan, September 15-18, 2008. Proceedings*, pages 1–16, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[34] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium*, volume 13, pages 223–238, 2004.

[35] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. *HotCloud*, 9:3–3, 2009.

[36] A. Seshadri, M. Luk, and A. Perrig. SAKE: Software attestation for key establishment in sensor networks. In *Proceedings of the 2008 International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2008.

[37] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Scuba: Secure code update by attestation in sensor networks. In *WiSe '06: Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94, New York, NY, USA, 2006. ACM.

[38] 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. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 1–16, New York, NY, USA, 2005. ACM.

[39] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272–282. IEEE, 2004.

[40] D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Trans. Inf. Syst. Secur.*, 3(1):51–62, Feb. 2000.

[41] G. E. Suh, C. W. Fletcher, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Author retrospective AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 68–70, 2014.

[42] G. Tan, Y. Chen, and M. H. Jakubowski. Delayed and controlled failures in tamper-resistant software. In *International Workshop on Information Hiding*, pages 216–231. Springer, 2006.

[43] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[44] X. Zhang and R. Gupta. Hiding program slices for software security. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 325–336, Washington, DC, USA, 2003. IEEE Computer Society.