FPGA based remote code integrity verification of programs in distributed embedded systems

Cataldo Basile, *Member, IEEE,* Stefano Di Carlo, *Member, IEEE,* and Alberto Scionti,

***Abstract*—The explosive growth of networked embedded systems makes ubiquitous and pervasive computing a reality. However, there are still a number of new challenges to its widespread adoption that include scalability, availability, and, especially, security of software. Among the different challenges in software security, the problem of remote code integrity verification is still waiting for efficient solutions. This paper proposes the use of reconfigurable computing to build a consistent architecture for generating attestations (proofs) of code integrity for an executing program, and for delivering them to the designated verification entity. Remote dynamic update of reconfigurable devices is also exploited to increase the complexity of mounting attacks in a real-word environment. The proposed solution perfectly fits embedded devices that are nowadays commonly equipped with reconfigurable hardware components exploited for solving different computational problems.**

***Index Terms*—Software protection, embedded systems, reconfigurable computing, dynamic update.**

Manuscript ACCEPTED version copy

I. INTRODUCTION

The steady proliferation of embedded systems from low-end to high-end devices has created demand for increased scalability, availability, integrity, and security of embedded software [1]. In particular, the ubiquitous use of critical embedded network applications makes security issues a serious concern. The common assumptions that embedded software does not represent not the prime target for tampering and fraud activities, or that tampering may produce a limited or negligible impact on the overall business do not hold any more [2].

The continuous improvement of cryptographic algorithms already provides a robust base to secure network communications. However, this offers only a partial protection as long as the communicating end-points still suffer from security issues. An increasing number of applications is therefore demanding for remote verification of software. Authorized entities must be able to verify if programs running on remote untrusted devices have been tampered with by malicious users.

Remote verification of programs includes verification of both the executed code and the execution state. This paper concentrates on the first of these two challenges referred to as the *remote code integrity verification* problem. Attacks that corrupt the functionality of the program by altering the sole execution state (e.g., program's variables) are out of the scope of this work. Protection mechanisms addressing these threats such as the one proposed in [3] may however gain advantages when code integrity has been properly established. Detecting modifications of the executed code does not necessarily imply a successful attack but, represents the evidence of a suspicious anomaly.

This paper exploits reconfigurable computing to build a secure architecture for the generation and the delivery of remote code integrity attestations (proofs) of an executing program. Reconfigurable computing creates a distinction between physical and logical hardware resorting to devices embedding Field Programmable Gate Arrays (FPGA). Since FPGAs provide a useful balance among performances, rapid time to market and flexibility, they have become the primary source of computation in many critical embedded applications [4]. In fact, in 2005 alone, an estimated 80,000 different commercial FPGA design projects have been initiated [5].

This paper introduces some important contributions. It proposes the use of FPGAs as a core of trust to securely compute code integrity attestations based on memory checksums and proposes a secure protocol do deliver them to a designated verification entity. The use of FPGAs in the attestation process is a key element to defeat a wide range of software integrity attacks as better detailed in section V. Existing techniques such as bitstream encryption and hardware obfuscation are applied to secure logical hardware components devoted to the software protection [6], [7]. Coupled with these techniques, the paper introduces an extended reconfigurable computing architecture allowing secure dynamic and continuos remote update of the hardware mapped into the FPGA. Dynamic update of logical hardware components reduces the time window to perform an attack and the time frame in which a successful attack remains in action, thus limiting the possibility of massive spreading of tampered applications. Reconfigurable computing coupled with dynamic hardware updates opens up a range of new opportunities for distributed embedded systems, including the possibility of achieving adaptability, functionality extension, and security [8]. It is worth mentioning here that the proposed solution does not provide a provable software security schema. The goal of this paper is a protection mechanism able to turn attacks based on programs' code modifications "exceedingly difficult to replicate in a real-world environment". The contributions of this paper are the result of three years of research on software protection techniques in the framework of the european project RE-TRUST (Remote EnTrusting by RUntime Software auThentication) [9] coupled with the expertise of the authors in the development of FPGA based self-reconfigurable systems for safety-critical applications [10], [11].

The paper is organized as follows: section II overviews related works on software protection while section III intro-

C. Basile, S. Di Carlo, and A. Scionti are with the Department of Control and Computer Engineering of Politecnico di Torino, Corso Duca degli Abruzzi 24 -10129 Torino, Italy. E-mail: {firstname.lastname}@polito.it.

This work was supported by the European Commission through the IST Programme under Contract IST-021186 RE-TRUST

duces the target scenario. Section IV details the protection schema whose security is analyzed V. Section VI evaluate the proposed method on a selected test-case and finally section VII summarizes the main contributions of the paper.

II. Related works

Software protection against tampering is not a new topic in the scientific community. Basically, software-based approaches are divided into local and remote solutions.

Local techniques build tamper proof methods directly into the program. Once deployed, the program is under full control of the attacker, and the protection solely relies on the strength of its implementation.

Among local techniques obfuscation [12] attempts to prevent reverse engineering by making it hard to understand the behavior of a program through static or dynamic analysis. Obfuscation is a widely studied topic. Collberg et al. [12] and Wang et al. [13] presented classes of transformations to confuse the control flow analysis of a program. Dalla Preda et al. [14] propose a formal framework for code obfuscation analysis based on abstract interpretation and program semantics. Theoretical works on white box computation [15], or layout randomization [16], [17] can also be mentioned in this field. However, negative theoretical results limit the possibility of building perfect obfuscators even for common classes of programs [18].

Integrity checking achieves tamper proofing by means of tamper-detection to identify program's modifications, and tamper-response to take actions when tampering is detected. Tamper-detection is usually based on check-summing techniques [19], [20], [12]. One of the main drawbacks of these methods is that they have been proven vulnerable to memory copy attacks [21]. Solutions to this attack include insertion of a large number of guards [22], computation of a fingerprint of the application code on the basis of its actual execution [23], or even the use self-modifying code techniques [24]. Control-flow integrity checking is also a promising technique to monitor the integrity of the execution of an application [25]. However, these approaches do not scale properly in real software setups. Moreover, even if strong local tamper-detection is achieved, dealing with tamper-response is still a challenge [26]. In fact, once an anomaly is discovered the use of the program must be interdicted. However, the code devoted to reactions can be easily identified and defeated.

Remote verification can overcome some of the drawbacks of local techniques. Time measuring is commonly exploited to detect attacks. It assumes that modification of the program inevitably increases its the execution time. Genuinity [27] explores the problem of detecting software running into a remote simulated hardware environment. SWATT [28] performs attestation on embedded devices with a simple CPU architecture. It uses a software verification function built in such a way that any attempt to tamper with it increases the computation time. Similarly, [29] compares the time required to run specific functions whose execution time is optimal, with a pre-computed execution time. Freshness is provided by a random challenge. The main problem of these approaches is that they rely on a trustworthy information about the hardware running the target application. This assumption is very strong and, even if satisfied, is vulnerable to proxy attacks. Moreover, to allow time accuracy, the system must stop all its activities during the measurement phase, thus making run-time verification impossible.

Whenever pure software solutions fail, trusted hardware modules can be exploited. Several solutions based on the Trusted Platform Module (TPM) [30], [2], [31], [32] have been proposed. Intel also has an initiative for the development of a secure platform [33]. The main drawback is that these hardware components cannot be replaced in case of design errors or unforeseen security threats. Moreover, they are not widely installed on all types of platforms (e.g., embedded devices) and cannot be easily plugged in as external components.

Solutions based on hardware monitors have been also proposed. Wolff et al. [34] presents a vault architecture for embedded systems to prevent the insertion of software corruptions, while Arora et. al [35] proposes a systematic methodology to design application-specific hardware monitors for any given embedded program. However, proposed approaches mainly rely on modifications of the embedded processor architecture. These changes are usually out of the user's ability. Moreover, protections are application-specific and difficult to generalize for a wide class of applications. Finally, they mainly targets sources in which the attacker is an external entity with limited control on the system. This scenario is somehow restrictive with respect to the one addressed in this paper.

Software protection devices (dongles) can be also mentioned here. A dongle can increase the level of security of a software since they are controlled and protected by the security solution provider [36]. However, dongles are mainly applied to software licensing and copy protection. Commercial attempts to implement monitoring techniques into USB dongles with computation capability do exist (Gemalto UpTeq™). However, monitoring is limited by the reduced access to the main system resources provided by the USB key.

Recently, reconfigurable hardware is increasingly gaining attention for the development of security applications [37], [38]. However, the main use of these devices concerns the implementation of security primitives such as encryption/decryption cores rather than focusing on software protection against tampering.

III. Remote code integrity verification principles

Two actors play in the remote code integrity verification problem:

- the *defender*: an entity with business, legal or regulatory interest of protecting the integrity of a program P executed in a remote untrusted embedded system, and
- the *attacker*: an entity aimed at modifying the behavior of P against the defender's will by means of modifications of its executed code.

Our solution can be formulated around the architecture proposed in Fig. 1 that includes three main components: the *untrusted host* (U), the *trusted host* (T) and a *network* connection (e.g., the Internet or a local/corporate network).

ACCEPTED Manuscript version Copy

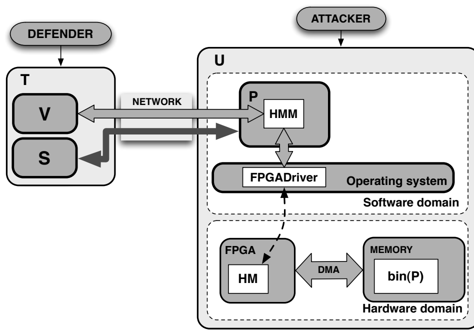


Fig. 1. Remote code integrity verification reference architecture: the untrusted host runs the target program under the full control of the attacker, while the trusted host verifies its integrity. The FPGA provides a core of trust in the untrusted environment of U.

U is a microprocessor based embedded device where P is executed under full control of the attacker. From the hardware standpoint (hardware domain of Fig. 1) the key requirement is the availability of a reconfigurable hardware block (FPGA). The FPGA can be either embedded into a complex System-on-Chip (SoC) or plugged through a standard socket (e.g., PCI, PCMCIA). Systems that are not natively provided with hardware reconfiguration facilities are therefore considered as well. The FPGA must have full access to the physical system's memory. This is however not a strong requirement. Direct memory access (DMA) is a common foundation of many high-performance devices easy to support in both embedded and plugged FPGAs. The FPGA hosts a soft hardware component referred to as *hardware monitor* (HM). It computes code integrity attestations for P exploiting the direct link between the FPGA and the system's memory (DMA in Fig. 1). HM implements a set of security mechanisms better discussed in the next sections of this paper, able to turn it into a core of trust in the untrusted environment of U. It can therefore be considered out of the attacker's control. A specific device driver ($\mathrm{FPGAdriver}$ of Fig. 1) allows access to HM from the software domain. A module called *hardware monitor manager* (HMM) embedded into P is responsible for collecting these attestations and delivering them to the designated verification entity. The security of this process is guaranteed by an ad hoc communication protocol (see section IV).

The trusted host is a trusted network node under the control of the defender. T is expected to deliver a certain service (S) to genuine programs, only. Modified applications should be detected and refused. A *verifier* (V) continuously checks the code integrity of the remote application by observing the attestations received from HM. Fig. 1 shows that direct communications are only possible between V/S and P through the network, and between P and HM through HMM and the $\mathrm{FPGAdriver}$. Communications between V and HM are therefore mediated by P and the operating system (OS) that might be under the control of the attacker. This in turn requires establishing a secure communication between V and HM.

A. Basic assumptions

The following specific assumptions are considered within this paper:

- The defender is free to operate any sort of modifications on P (while preserving its original functionalities) to make it difficult for the attacker to tamper with the code. Moreover, it knows the operating system U executes. The Linux operating system will be considered in this paper given the availability of detailed information on its internal organization.
- T is assumed intrinsically trusted by adopting both hardware and software protections (e.g., firewall) able to shield it against external attacks. The adversary can inspect its dynamic behaviors (e.g., the I/O from T) by interacting with it in a black-box manner, only.
- P must be a native code application whose executed code ($\mathrm{bin}(P)$ of Fig. 1) is loaded in the system's memory and visible to HM. Programs coded with intermediate languages and executed inside a virtual machine are therefore out of the scope of this work.
- P must be a network based application. It cannot execute without exchanging messages with S. Stand-alone programs that run without communicating on the network are out of the scope of this paper. Even if the attacker can carry out his malicious activity at any time, i.e., off-line modifications of the code completed prior to the execution as well as run-time modifications mounted while the program is executing, the need of continuous communication with S severely limits its power in analyzing the application by reducing the possibility of performing off-line testing with arbitrary inputs.
- Protected applications are not intended for high security domains (e.g., military, government). The attacker has no motivations to perform invasive hardware attacks to break the secrets contained in the FPGA (e.g., by means of electron microscopes). These attacks would be too expensive if compared to the advantages deriving from the misuse of the application.
- The remote attestation mechanism is built on top of secure state-of-the-art cryptographic algorithms. The adversary will be therefore unable to exploit any weaknesses in these methods to break the remote attestation process.
- The attacker wants to obtain the service S provided by T, he thus has no interest in performing Denial of Service (DoS) attacks against the remote attestation procedure. It would be meaningless for the attacker to avoid attestation requests from V or to avoid answering these requests. This would be immediately identified as an attack thus resulting in a cut of the deployed service.

IV. Computing code integrity attestations

The steps to request, compute, deliver and verify code integrity attestations based on the architecture of Fig. 1 are summarized in the remote attestation protocol of Fig. 2.

Basic elements of the protocol are: a session key K_S shared between HM and V (valid at most t seconds), a secure one-way function H (e.g., a cryptographic hash SHA-1) used to calculate a Hash-based Message Authentication Code (HMAC) and a symmetric block cipher E_{K_s}/D_{K_s} (e.g., AES). Furthermore, V and HM share a request counter i. The way the

ACCEPTED Manuscript version copy

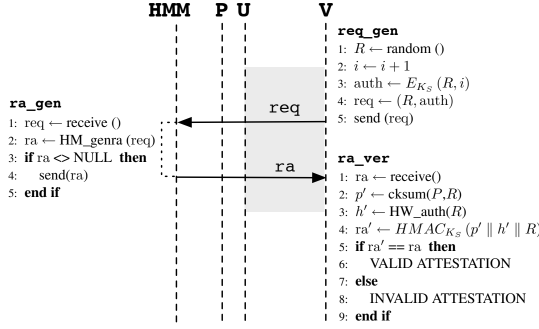


Fig. 2. Remote attestation protocol. For the sake of readability, retransmissions in case of missing responses are not depicted.

Algorithm 1 Compute attestation

Require: req,K_S,i
1: $(R, \mathrm{auth}) \leftarrow \mathit{split}(\mathrm{req})$
2: $(R', i') \leftarrow D_{K_S}(\mathrm{auth})$
3: **if** $R' = R$ and $i' > i$ **then**
4: $\quad i \leftarrow i'$
5: $\quad p \leftarrow \mathrm{cksum}(P, R)$
6: $\quad h \leftarrow \mathrm{bs_auth}(R)$
7: $\quad \mathrm{ra} = \mathit{HMAC}_{K_S}\,(p \parallel h \parallel R)$
8: $\quad$ return (ra)
9: **else**
10: $\quad$ return (NULL)
11: **end if**

session key K_S and the initial request counter i are exchanged will be discussed in section IV-A.

V initiates the protocol generating a remote attestation request req (req_gen phase in Fig. 2). V is the only actor allowed to perform this operation; it will discard any connection not following the proper message flow. req includes a random number R (req_gen-step 1, Fig. 2) and a request counter i properly incremented at each request (req_gen-step 2, Fig. 2), both encrypted with the shared session key, i.e., $\mathrm{auth} = E_{K_S}\,(R, i)$ (req_gen-step 3, Fig. 2). R is used as a nonce to avoid reply attacks against remote attestations and to avoid that requests can be replied to perform DoS against HM (req_gen-step 4, Fig. 2). The minimum suggested length for auth is the size of the block of the selected cipher (e.g., 128 or 256 bit). With a block of 128 bit a possible choice is 96 bits for R and 32 bits for i. This makes also very unlikely the repetition of random numbers within the session key validity time t. Once the request is ready, V opens an authenticated channel with P that is listening for connections, and sends $\mathrm{req} = (R, \mathrm{auth})$ (req_gen-step 5, Fig. 2). Since the attacker has full control on U, the authenticated channel is effective up to the untrusted host, only.

When P, or more precisely HMM, receives a remote attestation request (ra_gen-step 1, Fig. 2), it forwards the request to HM (ra_gen-step 2, Fig. 2). HMM does not take any active action in the remote attestation generation process. Communication with HM is achieved by means of the HM_genra(req) system call exported by the FPGAdriver (Fig. 1) and therefore contained inside OS. Together with req, this function also provides HM with the entry point of the page table of P required to correctly identify the process in the physical memory.

Whenever HM receives a new request, it computes a new remote attestation ra according to Alg. 1 and returns it to HMM. Finally, HMM forwards it to V, and the channel is immediately closed (ra_gen-step 4, Fig. 2).

To compute ra, HM uses its copy of the session key to extract the random number and the sequence counter (R', i') (Alg. 1 - step 1-2). The request is valid if $R' = R$ and, as a reply protection, if $i' > i$ (Alg. 1 - step 3). Consecutive indices are not mandatory since V will try with more than one request in case of missing responses. If this verification fails, HM simply ignores the request without further actions (Alg. 1 - step 10). Otherwise, HM updates its request counter (Alg. 1 - step 4) and collects a set of information p proving the integrity of P, and a set of information h proving its own integrity (Alg. 1 - step 5-6). It then calculates the HMAC of this information including the random number taken from the request (Alg. 1 - step 7, where $\parallel$ indicates the concatenation operation) and returns it to HMM. It is worth to remember here that HM is an hardware block configured into the FPGA. The operations of Alg. 1 are therefore executed by the hardware components it contains. According to section III and to what will be discussed in the remaining of this paper, the FPGA is configured in such a way to represent a root of trust into U, thus guaranteeing security in the computation of Alg. 1.

When V receives the response to its request (ra_ver-step 1, Fig. 2), it locally recovers the same integrity information provided by HM (p', and h') because it fully knows both P and HM (ra_ver-steps 2-3, Fig. 2). It calculates $\mathrm{ra}' = \mathit{HMAC}_{K_S}\,(p' \parallel h' \parallel R)$ (ra_ver-step 4, Fig. 2) and verifies if $\mathrm{ra}' = \mathrm{ra}$ (ra_ver-step 5, Fig. 2). If the remote attestation is invalid V tries again until reaching n_{req} consecutive requests. If all fail, it waits for a specific timeout and tries again with other n_{req} requests. If even this set of requests fail, it invalidates the current key and reacts according to a specific policy (e.g., by disconnecting U from the given service). $n_{req} = 3$, i.e., a total of 6 attempts, may represent a reasonable trade-off between security and toleration of network delays or loss.

V continuously logs all requests, taking track of those that failed. The analysis of log information can be exploited to identify suspicious users. A remote attestation is considered valid only if received within a given time t_{ra}.

A. Hardware monitor and key establishment protocol

The most security sensitive part of the remote attestation protocol is the possibility of securely sharing the session key K_S and the request counter i between V and HM. Remote dynamic update of the FPGA content is exploited to solve this problem. Together with providing an efficient method to exchange the requested secrets, dynamic update time constraints the activity of the attacker and limits the lifetime of an attack to the time frame between two updates.

Fig. 3 proposes a generic partitioning of the FPGA block of Fig. 1 including the set of elements required to implement HM and to correctly establish the shared secrets. All blocks

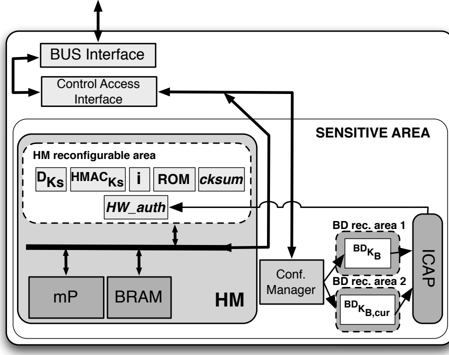


Fig. 3. FPGA partitioning including all blocks required to implement *HM* and to correctly establish the shared secrets

directly involved in the computation of ra are confined inside a so called *sensitive area*. The basic property of this area is that it is isolated in the FPGA with no direct connections to external pins. The only way to access its internal resources is through a set of predefined commands provided by the Control Access Interface.

HM is composed of a fixed part and a reconfigurable area. The fixed portion includes a very simple microprocessor (mP) providing basic memory access, mathematic, and comparison instructions, and a block of memory (BRAM). The reconfigurable area is a portion of the FPGA designed to be reconfigured at run-time while the system is performing its tasks. It is devoted to host the set of security cores required to implement Alg. 1, that is, D_{K_S}, cksum, bs_auth, $HMAC_{K_S}$, i, and a small ROM containing the program executed by mP. While D_{K_S} and $HMAC_{K_S}$ implement state-of-the-art cryptographic algorithms, the way cksum and bs_auth work will be detailed in the next subsections. Dynamic reconfiguration of the FPGA allows V to prepare a bitstream containing the required secrets, denoted as $bs\,(D_{K_S}, i, HMAC_{K_S}, \mathrm{cksum}, \mathrm{bs_auth})$, and to place it in the *HM* reconfigurable area through *HMM*. This operation resorts to a configuration manager block and to an Internal Configuration Access Port (ICAP) commonly available in contemporary FPGAs (Fig. 3).

The security of this remote update mechanism strongly depends on the complexity of full FPGA's bitstream reversal. Bitstream reversal is defined as the reverse engineering process that turns an encoded bitstream into a functionally equivalent description of the original design (e.g., netlist, hardware description language) [39]. Full bitstream reversal would, of course, reveal the FPGA design. Keys and security primitives hidden in the bitstream would therefore be compromised. As will be better discussed in section V, full FPGA's bitstream reversal is generally known to be a non-trivial problem. Nevertheless, the complexity of this task can be further increased by applying additional techniques.

Encryption can be applied to obtain bitstream confidentiality. At each remote update the bitstream is encrypted with a symmetric key K_B. The FPGA performs the reverse operation decrypting the incoming bitstream and recreating the intended configuration. This operation is performed by a *bitstream decryptor* (*BD*) block placed in the FPGA (BD_{K_B} and $BD_{K_{B,cur}}$ of Fig. 3).

Together with bitstream encryption, netlist obfuscation is exploited to increase the complexity of understanding security cores contained in a bitstream. Obfuscation is increasingly exploited in hardware design for Intellectual Property (IP) protection of hardware cores [6], [7]. Solutions include Hardware Description Language (HDL) level obfuscation [7] as well as netlist level obfuscation [6]. In particular, Chakraborty et al. [6] proposes an interesting technique to provide obfuscation of netlists together with a metric to evaluate the effects of the design modifications on the resulting obfuscation. Commercial solutions are also available (Helion Technology [40]). All these solutions are exploited in this paper as an instrument to provide additional level of security to *HM*, and therefore to establish the root of trust into the FPGA first mentioned in section III. Different types of obfuscations can be combined together and the complexity of the obfuscated cores can be additionally increased by including bogus hardware, or by using different routing schemes into the device. We denote with $\mathcal{O}\,(B_1, B_2, \ldots, B_n)$ the application of a set of obfuscation transformations to a set of hardware blocks B_1, B_2, ..., B_n. The obfuscation transformations are in general not the same for each block, nor in two applications of the $\mathcal{O}$ function.

Being the FPGA not provided with a factory certified endorsement key that can be used to perform bitstream encryption and therefore secure remote updates, an ad-hock key establishment protocol is introduced (Fig. 4).

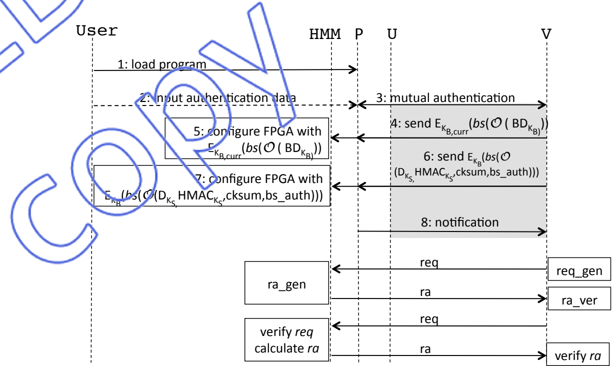


Fig. 4. The key establishment protocol

When P is executed (Fig. 4, step 1) the user authenticates to V (Fig. 4, steps 2-3). A strong authentication method is required. It must produce an authentication key shared between P and V used to establish an authenticated channel conveying all following messages (e.g., Needham-Schroeder-Lowe protocol [41]). Even in this case the channel is considered secure up to U, only. The user's authentication is performed in T; hence, the attacker cannot take advantage from controlling U.

The first step to perform is to correctly establish the bitstream encryption key. V prepares a bitstream decryptor *BD* with a bitstream symmetric encryption key K_B, applies a set of obfuscation transformation on it, encrypts the corresponding bitstream using the last agreed bitstream symmetric encryption key $K_{B,\mathrm{curr}}$, and sends it to P through the secure channel (Fig. 4, step 4). P receives the entire bitstream and configures it into the FPGA through *HMM* (Fig. 4, step 5). The bitstream

Manuscript ACCEPTED version COPY

is placed in one of two available reconfigurable areas (BD rec. area 1 or BD rec area 2 of Fig. 3). The reconfigurable area storing the current bitstream decryptor ($BD_{K_{B,curr}}$ of Fig. 3) is used to decrypt the new bitstream and to place it in the second area through the ICAP. The two areas are therefore used alternatively to host the current BD.

V then prepares a bitstream containing the security cores required to compute Alg. 1 ($bs\left(\mathcal{O}\left(D_{K_S}, i, HMAC_{K_S}, \text{cksum}, \text{bs_auth}\right)\right)$), encrypts it with the bitstream symmetric encryption key K_B, and sends it to P through the secure channel (Fig. 4, step 6). P receives the entire bitstream and configures it into the HM reconfigurable area (Fig. 3) through HMM (Fig. 4, step 7).

V expects an acknowledgement, within a given time, of the termination of the configuration phase to start the remote attestation protocol and to update the user's state information (Fig. 4, step 8).

The secrets exchanged through the key establishment protocol are considered valid only for a limited period of time. If the secrets expire or the request counter reaches its maximum, V starts the key establishment protocol. The case of the first connection of the user is analogous to a standard expiration case. The only difference is that there is no shared secret thus the BD sent during the first execution of the key establishment protocol is not encrypted.

B. Computing code integrity information

Code integrity information of P are produced by computing a checksum of the software code loaded in the system's memory. cksum (Fig. 3) is the block in charge of performing this computation. It reads (part of) the software code as an input and produces the corresponding checksum. The random number R included in the remote attestation request is used as a seed to perform a pseudo-random walk through the program's memory. Several functions can be exploited to compute the checksum (e.g., SHA-1, MD5, etc.). These techniques constitute a pool of methods to create different hardware monitors.

Reading the program's memory requires to navigate the page table of the process to transform virtual addresses into physical memory regions. mP can assist cksum in performing this operation. Nevertheless, this introduces a main security problem since, as previously anticipated, the entry point of the process page table is provided by the operating system that is under the control of the attacker. OS should be therefore validated in order to trust this information. cksum, together with mP, should therefore perform the following steps:

1) computing a checksum of the text of the OS kernel including the FPGA driver, and
2) computing a checksum of the portion of P currently loaded in memory

The two checksums, together with the list of inspected memory pages represent the integrity information p used in Alg. 1. Computing checksums of the kernel text of an operating system to detect malicious modifications is not new. In particular [42] proposes a solution that perfectly fits our architecture. It relies on hardware-based RAM acquisition to check the integrity of a Linux operating system. A snapshot of the system's memory is sent through the bus (PCI in the case of the paper) to a co-processor in an isolated environment inaccessible to the compromised machine where the system's integrity is continuously checked based on MD5 checksums.

C. Computing FPGA configuration integrity information

Together with the integrity of P, the configuration of the full FPGA must be validated in order to prevent the use of compromised designs. The bs_auth block of Fig. 3 is responsible for computing a checksum h of the FPGA configuration memory. The FPGA configuration memory can be easily read back through the ICAP interface (Fig. 3). The computed checksum can be then included in ra to show the HM integrity. This approach is commonly used in remote update of FPGAs to control whether the configuration was successful or not. It can therefore be easily implemented in contemporary devices.

V. Security Analysis

Several aspects must be considered to analyze strengths and weaknesses of our remote attestation mechanism. They include security of used cryptographic primitives, security of protocols' phases in isolation and when combined together and protection against environmental attacks (e.g. hardware, operating system, etc.). While security of cryptographic primitives is guaranteed by the use of state-of-the-art secure cryptographic algorithms (e.g., AES and SHA-1), the security of the remaining aspects need to be carefully analyzed.

A. Analysis of protocols

If E_{K_S}/D_{K_S} and the one way function H used by $HMAC$ are implemented using secure cryptographic primitives (e.g. AES for E_{K_S}/D_{K_S}, and SHA-1 for H) and K_S is secret and it is shared between HM and V, proving that the remote attestation protocol guarantees: (i) integrity and authenticity of the attestation request from V and (ii) integrity and authenticity of the response from, HM is straightforward.

The key establishment protocol assures that V and HM share the same key K_S. When the protocol correctly terminates, the FPGA is configured with the right hardware monitor HM that embeds K_S inside D_{K_S} and $HMAC_{K_S}$ (see Fig. 3). In fact, the attacker does not have any interest in configuring the FPGA with wrong bitstreams.

V and HM perform well established authentication procedures: V uses unilateral symmetric authentication using random numbers to prove its identity to HM, and HM authenticates to V by performing unilateral authentication using a keyed one-way function [43]. The security therefore depends on the impossibility of breaking H and E_{K_S}/D_{K_S}. This is however guaranteed by our preliminary assumption that H and E_{K_S}/D_{K_S} are implemented using secure cryptographic primitives and that the attacker cannot therefore exploit their weaknesses to perform its malicious activity (see section III-A).

The integrity of P and HM is guaranteed since remote attestations are produced using a secure one-way function using

ACCEPTED Manuscript version copy

the signature of the program p and the FPGA configuration integrity information h. To forge valid attestations the attacker needs to obtain K_S, p, and h by either performing reverse engineering of the FPGA bitstream or other types of attacks to the FPGA whose feasibility will be discussed later.

The next subsections analyze a set of common attacks against the proposed protocols.

1) Reply attacks against remote attestations: The random number R included in auth is used as a nonce to avoid reply of previous valid remote attestations. The length of R (e.g., 96 bits) makes very unlikely the repetition of random numbers within the validity time t of K_S. The couple (R, i) is therefore never repeated during the secrets validity time. Due to the loose verification of the request counter i ((Alg. 1 - step 3)), R is sent both in clear and ciphered inside auth for verification purposes (Fig. 2).

2) Reply attacks against remote attestation requests: The usage of the request counter i avoids that remote attestation requests can be replied (Alg. 1 - step 3).

3) Cryptanalysis attacks against K_S: Remote attestation requests protection against reply, also protects from cryptanalysis attacks against K_S. In fact, since HM avoids to produce the attestation in case of invalid requests, it cannot be forced to provide information about K_S by flooding it with fake requests.

To obtain information about K_S an attacker is therefore forced to create valid requests. However, forging valid requests without knowing K_S and the request counter i is a computationally heavy task. A brute force attack to find (R_f, auth_f) such that $D_{K_S}(\mathrm{auth}_f) = (R'_f, i_f)$ with $i_f > i$ and $R'_f = R_f$ is exponential in the minimum between the length of the block and the key of the used encryption algorithm. To avoid cryptanalysis attacks, the remote attestation protocol must not use stream encryption algorithms. In fact, these algorithms would permit to split auth into the encrypted part of R and the encrypted part of i, thus making the generation of valid requests easier.

4) Man in the Middle attacks: We consider here for the first time Man in the Middle (MITM) attacks. The MITM controls the network. He can perform passive attacks (eavesdropping), active attacks (packets modifications), and may be interested in performing DoS attacks. The goal of the MITM is different from the attacker's goals presented in section III. It does not have any personal advantage from using tampered programs even if he is not explicitly against it. The attacker considered in this paper is actually an allied in countering MITM attacks. In fact, it needs the service provided by V to attest the integrity of P. It will actually protect U and will not allow third parties to access or make HM unavailable.

MITM attacks are avoided by resorting to an authenticated channel between T and U. Opening an authenticated channel is a computationally expensive task whose impact is usually mitigated by techniques based on resume functionalities.

5) DoS against V: The fact that only V may initiate the remote attestation protocol reduces the vulnerability of V to DoS attacks, both from the attacker and the MITM. The MITM cannot perform DoS attacks against V because remote attestations are only transmitted inside the authenticated channel.

6) DoS against HM: By our assumptions, the attacker is not interested in performing DoS against himself (see section III-A). The MITM cannot perform DoS attacks against HM because attestation requests are transmitted inside the authenticated channel. The MITM can still block all messages (requests, attestations, bitstreams). Nevertheless, this vulnerability is a common threat with network applications.

7) Protocols verification: The security analysis produced so far is based on the consideration that the proposed remote attestation mechanism is built on top of secure cryptographic primitives and authentication algorithms. While this guarantees a strong base for the overall security we still need to verify that the composition of all primitives does not introduce threats. This has been possible using Scyther [44], a tool for the verification, the falsification, and the analysis of security protocols.[1]

Both the attacker proposed in section III operating on U, and the MITM have been considered.

The MITM has no control over T thanks to the authenticated channel between V and P. Hence, he cannot forge authentic messages to impersonate V. For this model the claimed properties that have been verified are: secrecy of the authentication keys exchanged with the authentication protocol (Needham-Schroeder-Lowe in our case), secrecy of the keys K_B and K_S, and correctness of the authentication phases. Scyther reported that *all claimed properties have been satisfied*.

The general attacker that has full control of U knows the secure channel authentication key and is therefore able to impersonate V. Moreover, there is not an authentication process between P and HM. In this case the claimed property that has been verified is the secrecy of both K_S and K_B. Scyther reported the existence of *exactly* five different patterns (attacker's behaviors) that may represent a security problem. Nevertheless, a deeper analysis highlighted that:

- the first pattern represents the correct behavior. Notice that the attacker may send to V a fake notification before the FPGA has been completely configured with the hardware monitor, but this does not affect the secrecy;
- the other four patterns have a common property: the attacker may force an invalid key into the FPGA by creating fake monitors. This does not represent a real threat. In fact, the use of the wrong key will be immediately detected by V. In our model, in case of $2n_{req}$ failed remote attestations V reacts according to a policy, e.g., it may block the usage of the program. In conclusion, the attacker may only perform DoS to himself and the FPGA, but this is in contrast with our assumptions.

B. Complexity of bitstream reversal

A key element to guarantee the security of the proposed remote attestation procedure is the inability of performing full FPGA's bitstream reversal of both HM and BDs (see section IV-A). This allows to turn these components into roots of trust into U.

[1]This paper has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the authors. This includes all tools and code used to perform formal verification of the different protocols.

ACCEPTED Manuscript version copy

The obscurity, complexity, and size of FPGA bitstreams confirm the general assumption that bitstream reversal is difficult and time consuming, though theoretically possible. FPGA's bitstream encoding is largely undocumented and obscure, even if not encrypted or confidential in a cryptographic sense. FPGA vendors keep this encoding a secret as they do for the chip's own design and layout information. Furthermore, the most important processing step in the bitstream generation process, i.e., place and route, is a very complex problem implemented in a non deterministic way. This makes it difficult to reliably generate slightly altered bitstreams and observe incremental changes in order to infer the function of the altered locations [45]. In fact, there are no reports of successful reversal of modern FPGA bitstreams as defined above or even a cost estimate that is backed up by data and empirical analysis [46], [47], [39], [48], [49]. In the 1990s the NeoCad company reverse engineered Xilinx's bitstream generation software, not the bitstream itself. Clear Logic was able to use Altera's software-generated bitstreams to produce laser-programmable circuits. Although both cases involved the generation of compatible bitstreams, neither company was able to completely reverse engineer them to obtain the original functional description. We must also remember that following the Moore's law, compared to today's FPGAs the ones in NeoCAD and Clear Logic's time were much less sophisticated.

Partial bit-stream reversal, which decodes static data from bitstreams, is far easier [45], [50]. Hiding cryptographic keys directly into look-up tables and RAMs must be therefore avoided, and the keys must be directly hardwired into HM.

Even if partial decompilation of the bitstream is possible, the textual representation obtained by these tools is not a true netlist. Furthermore, even assuming that the netlist stage could be reached, higher-level analysis tools would be required to actually structure and make sense of the data [45]. Since this difficulty is not exactly measurable as it happens for measurable cryptographic standards, additional levels of complexity have been exploited in this paper to increase the provided level of protection.

Netlist obfuscation makes even worse for the attacker the comprehension of a fully or partially reversed netlist.

Together with obfuscation, remote dynamic update of the FPGA gives the attacker a limited time to succeed in his action. Dynamic updates guarantee that, even if the attacker achieves its goal, it should start from scratch after each replacement (*time-limited successful attacks*). Moreover, this technique increases the difficulty of mounting a successful attack by creating a dependency of the current HM from all previously released BDs. Fig. 5 graphically shows the effect of the FPGA dynamic update on the attacker's activity. An attacker must first reverse engineer $BD_{K_{Bi}}$ to extract K_{Bi}. Only when K_{Bi} is extracted he may obtain in clear the bitstream of the current HM_i and, by reverse engineering also this component, he may try to deduce K_{Si}, p, and h. The time the attacker has to perform these operations is limited by the secrets expiration time t. This parameter can be regulated according to the security requirements, but it is reasonable to consider multiple of days or even weeks. If different versions of the monitors are generated including enough diversity, the time to mount an attack increases linearly with the number of replacements. This also reduces the possibility that discovering past keys would allow the attacker to compromise future session keys.

According to the above literature this gives a decent level of assurance that FPGA bitstreams cannot easily be reverse engineered and that direct modifications of the logic are difficult.

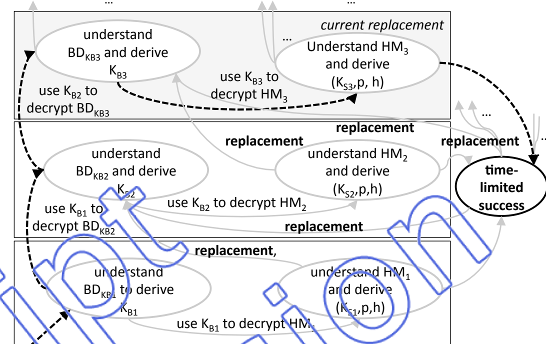


Fig. 5. Effect of the FPGA dynamic update on the attacker's activity

Remote update of the FPGA with several versions of equivalent HMs and BDs also provides protection against massive attacks. Using a single hardware monitor would expose programs to potential mass deployment of attacks. In fact, an attacker succeeding in understanding the hardware monitor could distribute to a large audience a tampered version of the program able to circumvent the implemented protection technique. By updating HM the attacker must reverse engineer the full library of available monitors to generate an (almost) effective crack. Due to the complexity of the bitstream reversal process, this time can be considered bigger than the usual lifetime of a program.

C. Environmental attacks

This section analyzes a set of attacks that exploits the running environment to be mounted.

1) Memory copy attacks: The memory copy attack consists of a fake paging table provided to HM in order to force it to create the attestation on a memory area containing a genuine copy of P, while a tampered running copy is placed in a different area [21]. Mounting memory copy attacks without compromising the kernel of the operating system that is monitored by HM is hard to achieve, thus guaranteeing a reasonable protection against this type of threats.

2) Bus resets: By trapping in a cycle of resets the bus where the FPGA is connected, it is possible to prevent HM, and in general all devices connected to the same bus, from performing their tasks. This is equivalent to perform a DoS attack which is against the interest of the attacker (see section III).

3) Power-off attacks and FPGA reconfigurations attacks: As introduced in section V-B, to decipher the bistream of HM the attacker needs to recover all previous bistream encryption keys K_B by reverse engineering the corresponding BD (see Fig. 5). The current BD can be easily maintained in the FPGA even in case of loss of primary power. SRAM-based FPGAs

ACCEPTED Manuscript version copy

are manufactured using a high performance low-power CMOS process. They can preserve the configuration data stored in the internal static memory cells during a loss of primary power by forcing the device into a low-power non-operational state, while supplying a minimal current from a battery.

An attacker may try to reduce the number of BDs to understand by removing the back-up battery and therefore cleaning the FPGA. A similar situation can be created by configuring the FPGA with a bogus design preventing the correct configuration of the received updates. Both situations can be detected since the FPGA configuration cannot properly complete and the corresponding notification cannot be sent (see Fig 4).

Failing a remote update can be immediately considered as an attack, or may be addressed by cleaning the FPGA and starting a first instance of the key establishment protocol with a not-encrypted BD (see section IV-A) thus reducing the number of BDs the attacker need to understand to perform its activity. These events must be logged by V to identify suspicious situations to manage based on a specific security policy.

According to section V-B this does not represent a major security problem for our protection technique. Nevertheless, if not addressed it may contribute to reduce the strength of the overall protection.

4) Timing attacks: If an attacker could foresee when remote attestation requests are sent by V, he might develop a clever crack able to automatically modify and repair the memory area checked by HM. In alternative, U may force the swap of tampered pages. This type of attack certainly deteriorates the performance of the program proportionally to the number of generated remote attestation requests. To prevent these attacks, the intervals between remote attestation requests are randomized to make their occurrence difficult to predict.

5) Relocation or cache based dynamic attacks: A very skilled attacker could try to maintain a consistent image of the monitored memory while hiding malicious code elsewhere, such as in the processor cache. In this situation, the malicious code would not be detected by HM.

However, it is currently unclear to what extent such attacks would succeed on a permanent scale. Caches get flushed frequently and more extensive attempts to relocate large portions of the operating system or page tables would likely require difficult changes to all running processes.

6) Parallel execution of multiple copies of the program (cloning attack): With cloning attack we refer to the possibility of running in parallel two copies of the program (P and P'). The tampered version of P' interacts with S to receive the given service while all attestation requests are forwarded to the untampered version P of the program that answers to V with the correct remote attestations. This mechanism must be implemented without any modification to the code of the original program P. This particular scenario is not covered by our solution and also by previous publications presented in section II. However two considerations reduce the criticality of this attack. First, even if theoretically possible, implementing the parallel execution of the two processes (P and P') with the related communication mechanisms without any modification to the code of P and to the operating system is not trivial. Second, this type of attack requires the possibility of running a clean (licensed) version of the software. This is somehow in contrast with the aim of several attackers that would like to run tampered unlicensed versions of the software. Running the clean licensed version of the program may also lead to the identification of the identity of the client which is obviously against the attacker's will.

VI. Case study and evaluations

The goal of the experimental setup of this section is to perform feasibility, scalability and performance analysis on the proposed methods. CarRace is a small sized client-server network game written in C++ designed for small terminals (e.g., mobile terminals) running the Linux operating system. The application allows players to connect to a central game server (about 4,500 lines of C++ code) using a game client application (about 7,400 lines of C++ code) and race cars against each other. CarRace has been developed in the framework of the RE-TRUST project [9] to demonstrate the application of remote software protection techniques. During the race, each client periodically sends data (e.g., car position, direction, etc.) to the server which then broadcasts the data to the other clients allowing them to render the game on their screens. To reduce computational load on the server side, a critical issue of contemporary on-line game applications, CarRace demands most of the computations to the client that is executed in an untrusted environment. Monitoring the client's code integrity is therefore mandatory.

Fig. 6 shows our experimental setup. The verifier (V) and the game server representing the service S of Fig. 1 are executed on two workstations both equipped with two Intel Xeon 2.66GHz CPUs and 16GB of physical RAM. Both workstations run Ubuntu-Linux with kernel 2.6.32. V and S are executed on two different machines to allow performance analysis of V in isolation. The untrusted host U where the game client representing the program P to protect runs is built on a Xilinx ML605 demo board implementing a LEON3 microprocessor running the MontaVista Linux operating system with kernel 2.4.26. The ML605 board is a development environment for embedded systems design featuring Xilinx Virtex-6 FPGAs. A 100BASE-T ethernet router connects the different nodes guaranteeing the required exchange of messages. A laptop running an X server is used as a graphical terminal overcoming the unavailability of a full featured graphic card into the development board. Due to the availability of a single ML605 board, S implements a computer emulated player allowing a single real user to play against the computer. All experiments presented in the next subsections have been performed by bringing all systems to a known, minimal state. Only those processes required for the experiments have been preserved disabling typical operating systems' activities such as cron, syslog,or any other unnecessary service.

A. Verifier implementation and performance analysis

V implements a process that continuously waits on a TLS socket for new clients. When a new client connects, it forks generating a process to manage the new player. Two time-out

Manuscript ACCEPTED version Copy

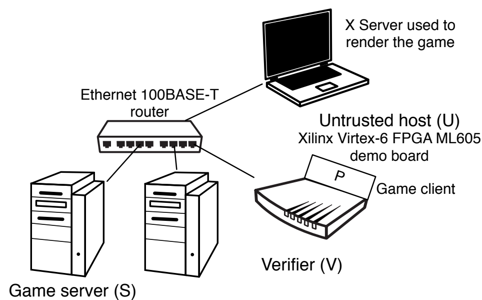


Fig. 6. Testcase architecture setup

for each client govern the rate at which remote attestations and key establishments are performed. A repository of six precomputed HMs and BDs allows to simulate the remote update process. V is highly configurable. Specific information on the program to monitor are all coded into a set of configuration files. It can therefore be reused to protect different programs. AES-128 is used as cipher to encrypt remote attestation requests, while SHA-1 is used as digest algorithm for the HMAC (see section IV). Implementations of security primitives exploit openssl libraries. This trivial implementation has opportunities for several improvements, e.g., a pool of pre-forked processes can manage different clients and remote attestations can be pre-computed when there are free resources. However it is enough to provide a preliminary assessment of the scalability of the proposed method.

1) Remote attestation performance analysis: Performances of V when executing remote attestations have been measured running a set of simulations with a variable number of served game clients (10, 100, 1000, or 10,000 clients) and a variable interval between remote attestations (240sec, 120sec, 60sec, and 30sec). A dummy client emulating the sole remote attestation process is used to perform measurements without need of an elevated number of demo boards and players. An additional dedicated workstation with the same characteristic of the one implementing V is introduced in the setup of Fig. 6 to execute all instances of the dummy client.

Measurements for each configuration have been collected by observing V for a window of 4 hours of continuos activity. Fig. 7 summarizes the obtained results.

Fig. 7(a) presents the average time to complete a full remote attestation for a given client (time spent to wait for ra from the client is not considered in the statistic). The average time does not show significant variations among the considered configurations even with the highest number of clients (values ranging between 34.3msec to 38.9msec). The low computation time guarantees the possibility of managing remote attestations at the desired rate even in the worst configuration (10,000 clients with 30sec between two requests, a very high and probably not realistic rate) thus providing a good scalability. The coefficient of variation (CV) computed for the different configurations show that measures are quite stable when the number of clients is not high ($\leq 1,000$) while their variability starts to be significant when considering 10,000 clients. While this is a sign that the load of the server is increasing, looking at the average computation time of measures performed with 10,000 clients (30sec, 60sec, 120sec, and 240sec) the small variations suggest that the size of CV is probably mainly influenced by the elevated number of generated processes (one for each client) rather then by the actual number of computed remote attestations. This is a limitation of our simple implementation that can be mitigated by implementing some of the optimizations mentioned above.

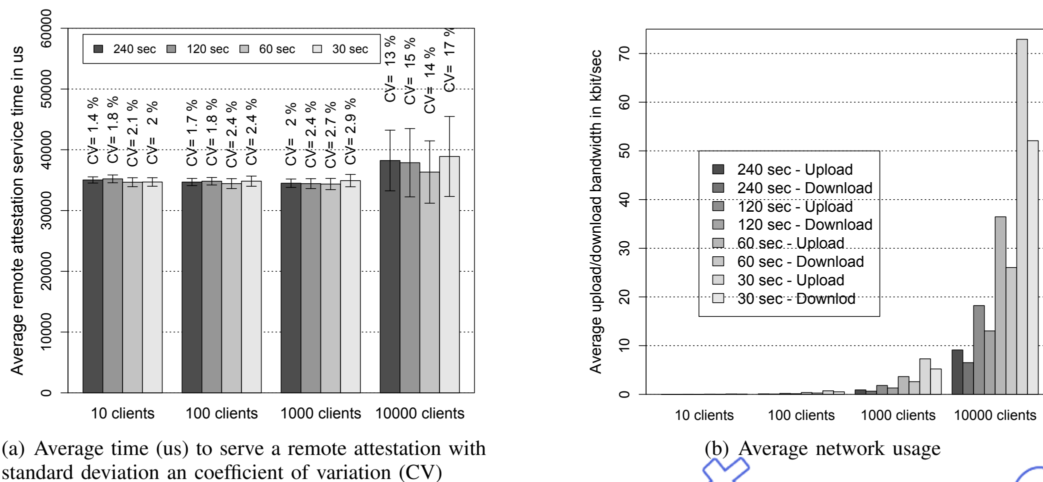
Consumed network resources are also negligible. The cipher used to generate remote attestation requests (AES-128) has a block size of 128bit. According to section IV our implementation uses a sequence counter i of 32bit and a random nonce R of 96bit. Considering that the nonce is also sent in clear the size of each request is 28byte. The size of the remote attestation returned by the client considering the use of SHA-1 as digest algorithm to compute the HMAC is 20byte. Fig. 7(b) shows the average upload/download network bandwidth consumed by V during the 4 hours of observed activities. As expected, the average bandwidth is directly related to the number of performed remote attestations. The request is anyway really low (a few tens of Kbits/sec) and quite leveled over the observation time (no significant peaks of use have been observed).

2) Key agreement performance analysis: The key establishment procedure does not represent a critical activity. The rate at which this process is performed is not high, we can expect multiples of days or weeks. Considering an implementation of V based on a library of precomputed HMs and BDs the majority of the computation time for the key establishment procedure is devoted to the bitstreams encryption. Our proposed experimental setup allows to perform encryption with AES-128 at a measured average rate of 138 MB/s (using blocks of 1024 byte). Considering a total average size of the bitstreams representing HMs and BDs of $\sim$ 3 MB, approximatively 46 bitstreams/sec can be encrypted, guaranteeing a good level of scalability even when no cryptographic accelerators are used. Network use is also negligible given the low rate at which this process is scheduled.

B. Untrusted host and HM implementation

The ML605 Xilinx board including a Virtex-6 FPGA implements the full untrusted host (Fig. 8). The system is based on the LEON 3 processor and uses the Advanced Microcontroller Bus Architecture (AMBA) as backbone. It includes 512 MB DDR SDRAM, an ethernet controller, and 64 MB of flash memory used as storage device, all available on the demo board.

The LEON3 is a SPARC-V8 compliant RISC open source processor. Our implementation works at a clock frequency of 50 MHz. An internal memory management unit (MMU) translates virtual addresses into physical ones. The LEON3 adopts a four levels page table. The virtual address is partitioned in four sections: three levels of displacements and an offset in the final page table (see the SPARC-V8 specifications for further details [51]). The same translation mechanism has been implemented in the library of available HMs to explore the memory space of the target program. FPGA designs have been produced



(a) Average time (us) to serve a remote attestation with standard deviation an coefficient of variation (CV)

(b) Average network usage

Fig. 7. Experimental results obtained by observing the server for four hours of operations considering 8 different configurations

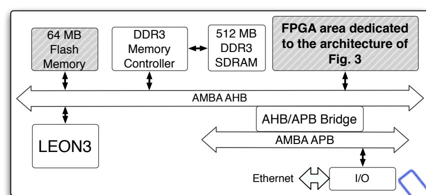


Fig. 8. Implementation of the untrusted host. The AMBA Advanced High-performance Bus (AHB) and the AMBA Advanced Peripheral Bus (APB) are used to connect the different cores.

using the Xilinx ISE environment. All bitstreams have been generated with the "Disable Readback" option. This prevents the possibility of reading back the FPGA configuration memory, while leaving full access to the ICAP. The system runs the MontaVista Linux operating system with kernel 2.4.26, with a custom driver implementing the FPGAdriver of Fig. 1.

The Xilinx Virtex-6 FPGA that equips the ML605 board, contains $37,680$ slices, 4 Ethernet MAC cores and 416 memory blocks of 36 Kb. Table I shows synthesis information in terms of slices required for the different structures. Results show how the size of HM is limited and can easily fit commercial devices.

TABLE I
SYNTHESIS RESOURCES UTILIZATION

Module	Slices	% of Virtex-6
U Main system (LEON3, AMBA, ...)	16,340	43.3%
HM (rec. area)	3,112	8.2%
HM (fixed portion)	2,910	7.7%

C. CareRace client performance analysis

A "remote attestation on" "remote attestation off" evaluation approach has been used to compare the impact of the remote attestation process on the CarRace game client. Similarly to the experiments performed for V, four intervals between remote attestations have been considered (240sec, 120sec, 60sec and 30sec). For each configuration 30 repetitions (car races) of the experiment have been performed.

The CarRace client implements an iterative process in which two sets of functions, i.e., core functions managing the game activity and rendering functions generating X commands to render the game on the console, are repeatedly executed. Table II presents averages of the total execution time of the two groups of functions (together with the corresponding CV to asses their dispersion) considering the 5 configurations (remote attestation off, and remote attestation on with the four considered intervals) and the 30 trials. The penalty column allows to evaluate the penalty introduced by the remote attestation process w.r.t. the execution without remote attestation. As expected, the more frequently remote attestations are requested by V, the more impact there is on the system's performances. While a very high remote attestation rate (30sec between two requests) produces a quite high overhead (18.31%), keeping the remote attestation interval between 60sec and 240sec allows to maintain the overhead at acceptable limits for the considered application (between 5.16% and 1.15%). Loss of performances are caused by the shared use of the bus/memory between the LEON3 and HM, by the time consumed by HMM (embedded into the protected program) and by the calls to the FPGAdriver, with the first contribution accounting for $\sim 20\%$ of the overhead. If required, custom hardware designs including multi-port memories can be exploited to mitigate this contribution.

Similarly to V the key establishment process does not represent a critical issue for the application. The average time to perform FPGA configuration with the proposed library of HMs is about ~ 1.5 s. This process can be executed in parallel with the execution of the program, and can be scheduled in the time interval between remote attestations in order to be almost transparent to the program's execution.

D. Simulated attacks

To asses the security of our selected test-case we performed an empirical experiment involving a group of seven PhD students of Politecnico di Torino attending a PhD course in

ACCEPTED Manuscript version Copy

TABLE II
COMPARISON OF THE CLIENT'S EXECUTION TIME WITH OR WITHOUT REMOTE ATTESTATIONS

	Rendering functions			Core functions		
	Avg	**CV**	**Penalty**	**Avg**	**CV**	**Penalty**
Remote attestation disabled	55.1068	3.02%	0.00%	18.9104	1.95%	0.00%
Remote attestation request every 240sec	55.4871	2.04%	0.69%	18.9974	2.04%	0.46%
Remote attestation request every 120 sec	55.8232	2.15%	1.30%	19.0749	2.99%	0.87%
Remote attestation request every 60 sec	56.8096	3.21%	3.09%	19.3018	3.21%	2.07%
Remote attestation request every 30 sec	61.1465	4.08%	10.96%	20.3003	3.90%	7.35%

Software Engineering. All students had at least two years of experience in research topics including software-engineering, security and FPGA design. A preliminary session was used to fully explain the characteristic of the system to attack and the goal of the experiment. All students were provided with the source code of the application, the documentation on the protection schema and a complete suite of tools (compiler, debuggers, FPGA design, etc.). They were asked to produce tampered versions of the CarRace client exploiting the attacks discussed in section V. To motivate the students, successful attacks were rewarded with a maximum score in the final exam. All student had one month to perform their activity.

The result of the experiment gave positive results. All tampered versions of the program (about 30 programs covering the range of attacks discussed section V), once installed and monitored have been correctly detected further confirming the security analysis proposed in section V. In particular, none of the tentatives of implementing the critical cloning attack was successfully completed. Additional empirical studies with larger groups of students and multiple repetitions are planned to provide a strongest empirical validation of the security of the system.

VII. CONCLUSION

This paper proposes the use of reconfigurable computing to securely implement remote code integrity verification of software including both protocols design and hardware architecture has been presented. One of the main contributions of the paper is the use of remote dynamic update of hardware components to make it difficult for an attacker to perform full bitstream reversal of hardware monitors thus defeating the protection technique. Experimental data on scalability issues give promising results.

Future activities include the development of a complete implementation of the framework able to consider the entire lifecycle of the program. This will also include fully automating the generation of different versions of HMs and BDs, the reduction of the overhead through a distributed set of trusted hosts, and the definition of a standard set of reactions to tampering detection. The production of the monitors is essentially an off-line process. Nevertheless, after the generation, they need to be catalogued and associated with the secrets they contain in order to be quickly available to V when needed. In a large scale scenario, the same agent may be sent to different users with different keys. Also distribution should be designed carefully to reduce coalitions.

REFERENCES

[1] F. M. Mendes Neto and P. F. Ribeiro Neto, *Designing Solutions-Based Ubiquitous and Pervasive Computing: New Issues and Trends*. Hershey, PA: Information Science Reference - Imprint of: IGI Publishing, 2010.

[2] D. Schellekens, B. Wyseur, and B. Preneel, "Remote attestation on legacy operating systems with trusted platform modules," *Science of Computer Programming*, vol. 74, no. 1-2, pp. 13 – 22, Dec. 2008.

[3] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," in *Proceedings of the Dependable Systems Networks Conference*, 2009, pp. 115 –124.

[4] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, June 2002.

[5] D. McGart, "Gartner dataquest analyst gives asic, fpga markets clean bill of healtg," *EE Times*, 13 June 2005. [Online]. Available: http://www.eetimes.com/news/latest/archive/?archiveDate506/18/2005

[6] R. Chakraborty and S. Bhunia, "HARPOON: An obfuscation-based soc design methodology for hardware protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, Oct. 2009.

[7] M. Brzozowski and V. Yarmolik, "Obfuscation as intellectual rights protection in VHDL language," in *Proceedings of the 6th International Conference on Computer Information Systems and Industrial Management Applications*, 2007, pp. 337–340.

[8] A. Benso, A. Cilardo, N. Mazzocca, L. Miclea, P. Prinetto, and E. Szilard, "Reconfigurable systems self-healing using mobile hardware agents," in *Proceedings of the IEEE International Test Conference*, . 2005, pp. pp.9–476.

[9] Re-trust (remote entrusting by run-time software authentication). [Online]. Available: http://www.re-trust.org

[10] S. Di Carlo, P. Prinetto, and A. Scionti, "A fpga-based reconfigurable software architecture for highly dependable systems," in *Proceedings of the IEEE Asian Test Symposium, 2009*, nov. 2009, pp. 125–130.

[11] S. Di Carlo, A. Miele, P. Prinetto, and A. Trapanese, "Microprocessor fault-tolerance via on-the-fly partial reconfiguration," in *Proceedings of the 15th IEEE European Test Symposium*, may. 2010, pp. 201–206.

[12] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, August 2002.

[13] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2001, pp. 193–202.

[14] M. Dalla Preda and R. Giacobazzi, "Semantics-based code obfuscation by abstract interpretation," *J. Comput. Secur.*, vol. 17, no. 6, pp. 855–908, 2009.

[15] H. E. Link and W. D. Neumann, "Clarifying obfuscation: Improving the security of White-Box DES," in *Proceedings of the International Conference on Information Technology*, 2005, pp. 679–684.

[16] M. Abadi and G. Plotkin, "On protection by layout randomization," *Computer Security Foundations Symposium, IEEE*, vol. 0, pp. 337–351, 2010.

[17] H. Xu and S. J. Chapin, "Address-space layout randomization using code islands," *J. Comput. Secur.*, vol. 17, no. 3, pp. 331–362, 2009.

[18] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2001, vol. 2139, pp. 1–18.

ACCEPTED Manuscript Version copy

[19] D. Aucsmith, "Tamper resistant software: an implementation," in *Proceedings of the First International Workshop on Information Hiding*, ser. Lecture Notes in Computer Science, R. Anderson, Ed. Springer Berlin / Heidelberg, 1996, vol. 1174, pp. 317–333.
[20] H. Chang and M. Atallah, "Protectioning software code by guards," in *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management*, 2001, pp. 160–175.
[21] G. Wurster, P. C. van Oorschot, and A. Somayaji, "A generic attack on checksumming-based software tamper resistance," in *In Proceedings of the IEEE Symposium on Security and Privacy*, ser. Lecture Notes in Computer Science. Springer, 2005, vol. 4437, pp. 127–138.
[22] J. Cappaert, B. Preneel, A. Bertrand, M. Matias, and D. B. Koen, "Towards tamper resistant code encryption: Practice and experience," in *Proceedings of the Second Information Security Practice and Experience Conference*, ser. Lecture Notes in Computer. Springer, 2008, vol. 4991, pp. 86–100.
[23] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. Jakubowski, "Oblivious hashing: A stealthy software integrity verification primitive," in *Proceedings of the International Workshop on Information Hiding*, ser. Lecture Notes in Computer Science, F. Petitcolas, Ed. Springer Berlin / Heidelberg, 2003, vol. 2578, pp. 400–414.
[24] J. T. Giffin, M. Christodorescu, and L. Kruger, "Strengthening Software Self-Checksumming via Self-Modifying Code," in *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005, pp. 23–32.
[25] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–40, 2009.
[26] G. Tan, Y. Chen, and M. H. Jakubowski, "Delayed and controlled failures in tamper-resistant software," in *Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems*, 2006, pp. 216–231.
[27] R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 295–310.
[28] A. Seshadri, A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: software-based attestation for embedded devices," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2004, pp. 272–282.
[29] 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 ACM Symposium on Operating Systems Principles*, 2005, pp. 1–16.
[30] T. C. Group. Trusted computing platform. [Online]. Available: http://www.trustedcomputing.org/
[31] P. van Oorschot, "Revisiting software protection," in *Proceedings of the Information Security Conference*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2851, pp. 1–13.
[32] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of the 3th USENIX Security Symposium*, 2004, pp. 16–16.
[33] Intel. Intel trusted execution technology. [Online]. Available: http://developer.intel.com/technology/security/index.htm
[34] F. Wolff, C. Papachristou, D. Weyer, and W. Clay, "Embedded system protection from software corruption," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, jun. 2010, pp. 223–229.
[35] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 12, pp. 1295 –1308, dec. 2006.
[36] T. Jiutao and L. Guoyuan, "Research of software protection," in *International Conference on Educational and Network Technology*, jun. 2010, pp. 410–413.
[37] O. Gelbar, P. Ott, B. Narahari, R. Simha, A. Choudhary, and J. Zambreno, "CODESSEAL: Compiler/FPGA approach to secure applications," in *Proceedings of the Intelligence and Security Informatics Conference*, 2005, vol. 3495, no. Lecture Notes in Computer Science, pp. 530–535.
[38] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, J.-P. Bossuet, and R. Vaslin, "Reconfigurable hardware for high-security/high-performance embedded systems: the SAFES perspective," *IEEE Trans. On Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, Feb. 2008.
[39] S. Drimer. Volatile fpga design security - a survay. [Online]. Available: http://www.cl.cam.ac.uk/~sd410/papers/fpga_security.pdf
[40] Helion technology. [Online]. Available: http://www.heliontech.com/
[41] G. Lowe, "Breaking and fixing the needham-schroeder public-key protocol using fdr," in *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer Verlag, 1996, vol. 1055, pp. 147–166.
[42] 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*, 2004, pp. 13–13.
[43] A. J. Menezes, van Paul C. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.
[44] C. J. Cremers, "The scyther tool: Verification, falsification, and analysis of security protocols," in *Proceedings of the 20th International Conference on Computer Aided Verification*, 2008, pp. 414–418.
[45] J.-B. Note and E. Rannaud, "From the bitstream to the netlist," in *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, 2008, pp. 264–264.
[46] T. Wollinger, J. Guajardo, and C. Paar, "Security on FPGAs: State-of-the-art implementations and attacks," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 534–574, 2004.
[47] S. Trimberger, "Trusted design in FPGAs," in *Proceedings of the 44th Annual Design Automation Conference*, 2007, pp. 5–8.
[48] D. Schellekens, P. Tuyls, and B. Preneel, "Embedded trusted computing with authenticated non-volatile memory," in *Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies*, 2008, pp. 60–74.
[49] T. Huffmire, B. Brotherton, T. Sherwood, R. Kastner, T. Levin, T. D. Nguyen, and C. Irvine, "Managing security in FPGA-based embedded systems," *IEEE Design and Test of Computers*, vol. 25, pp. 590–598, 2008.
[50] D. Ziener, S. Abmust, and J. Teich, "Identifying FPGA IP-cores based on lookup table content analysis," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2006, pp. 1–6.
[51] SPARC International Inc. The sparc architecture manual. [Online]. Available: www.sparc.com/standards/V8.pdf

ACCEPTED Manuscript Version copy