Verification of FPGA-augmented trusted computing mechanisms based on Applied Pi Calculus

Alessandro Cilardo and Andrea Primativo*

December 22, 2017

Abstract

Trusted computing technologies may play a key role for cloud security as they enable users to relax the trustworthiness assumptions about the provider that operates the physical cloud infrastructure. This work focuses on the possibility of embodying Field-Programmable Gate Array (FPGA) devices in cloud-based infrastructures, where they can benefit compute-intensive workloads like data compression, machine learning, data encoding, etc. The focus is on the implications for cloud applications with security requirements. We introduce a general architecture model of a CPU+FPGA platform pinpointing key roles and specific assumptions that are relevant for the trusted computing mechanisms and the associated security properties. In addition, we formally verified the proposed solution based on Applied Pi Calculus, a descendant of Pi Calculus, that introduces constructs allowing the symbolic description of cryptographic primitives. The verification phase was automated by means of ProVerif, a tool taking as input a model expressed in Applied Pi Calculus along with some queries and annotations that define security properties to be proved or denied. The results of the analysis confirmed that the properties defined in our work hold under the Dolev Yao attacker model.

1 Introduction and motivation

Information security has become a vital concern today and is expected to be even more important in the future. Security vulnerabilities have been successfully exploited in disparate types of systems, including personal communication [25], vehicles [19], and industrial appliances [15]. Cloud security, in particular, is attracting much interest in areas ranging from theoretical works on cryptography to architectural solutions and protocols, as major companies are migrating their services to the cloud for the reduced costs of outsourced computation. As observed in [23], data in the cloud can be accessed by the cloud provider, its sub-contractors, and employees. The service provider has indeed physical access to the machines and can easily carry out attacks, both at the software and physical level, to access user data. Outsourcing computation to the cloud needs, therefore, an additional level of trust. A key role in this scenario is played by trusted computing technologies that enable users to relax the trustworthiness assumptions about the provider operating the physical cloud infrastructure. For instance, Intel Software Guard Extensions (SGX) address the above issues by providing a mechanism to protect certain user memory regions, called enclaves, from processes running at higher privilege levels. Albeit recently introduced, SGX has already been addressed by a large body of research. [11] proposes a user credential protection architecture to be deployed in cloud environments. In [13] a verifiable and confidential implementation of MapReduce is built on top of SGX. Further relevant works addressing SGX include [3, 9, 3]. Notice that the above works are focused on performance and do not generally take a formal approach towards the verification of the security properties offered by the proposed solutions. On the other hand, there are a number of works involving formal verification of security properties [12, 7], some specifically addressing existing tusted computing solutions. For example [10] proposes an alternative approach to support concurrent isolated execution where the critical functions of attestation and memory isolation are delegated to a software monitor, whose correctness is showed by means of a machine checkable proof. [14] focuses on the formal modelling of SGX API in order to verify that an enclave satisfies confidentiality against an attacker accessing all the non-enclave memory. In particular, focusing

*The authors are with the University of Naples Federico II, via Claudio, 21 - 80125 Napoli, Italy, email: acilardo@unina.it

on a formal approach, the authors develop Moat, a system for statically verifying confidentiality properties of an enclave program.

Unlike the above works, this paper focuses on the possibility of embodying Field-Programmable Gate Array (FPGA) devices in trusted cloud-based infrastructures. FPGA acceleration has been explored for compute-intensive workloads like data compression, machine learning, data encoding and decoding in cloud and datacenter settings [26, 4, 16]. The main motivation behind our investigation is the potential role that FPGA acceleration may play for cloud-based applications where security is critical. In particular, we introduce a general architecture model of a CPU+FPGA platform pinpointing key roles and specific assumptions that are relevant for the trusted computing mechanisms and the associated security properties. In that respect, our work differs from the previous results as the definition of new architectures and mechanisms is carried out concurrently with the formal verification, which affected the way we shaped the proposed trusted computing solution. There are a number of application scenarios involving hardware acceleration in the cloud where security may play a crucial role. Some of them are listed below.

- Image processing that handles biometric information such as fingerprint images or clinical sensitive information, like radiography, can be successfully accelerated in hardware but, of course, biometric data and the radiography images should never be revealed to unauthorized parties.
- Data analytics can greatly benefit from hardware acceleration but, if the analysis is carried out on valuable data, there is the risk of heavy economical loss if the dataset, or part of it, is exposed to competitors.
- Cryptographic services, e.g. public key infrastructures, are another example of security sensitive applications often needing hardware acceleration, although the exchange of the encryption/decryption keys or other critical data between the processor and the accelerators may turn out to be a security bottleneck.

When such applications are outsourced to cloud services, the hardware acceleration platform is potentially exposed to malicious hands that have physical access to the device and can, hence, carry out invasive attacks accessing the data exchanged between the software environment and the accelerators. As a further concern, besides physical attacks, consider that a user application is not necessarily supposed to provide its own custom accelerator developed from scratch. Rather, it may require a third party accelerator. This introduces the problem of authenticating the third-party component, as an application should share sensitive data only with trusted accelerators. On the other hand, if the application wants to deploy a custom accelerator, it should be provided with guarantees that the reconfiguration bitstream remains confidential and the reconfiguration process is performed without any interference. The extended architecture model and trusted computing mechanisms introduced by this work directly address this challenge. The model fits platforms made of a general-purpose processor with trusted computing mechanisms augmented with a reconfigurable hardware fabric connected through a (possibly untrusted) channel eiher on- or off-chip. The key insight in the proposed solution is to retain the security properties already offered to software applications by the trusted platform and effectively extend the trusted computing base to the FPGA-implemented components. A protocol is proposed that enables a user application to securely reconfigure a custom accelerator, attest its instantiation on the reconfigurable fabric, and establish a secure communication channel with it. A possible technology mapping of such a model is represented by Intel SGX extended to the new Intel Xeon+FPGA platforms, for which no trusted mechanisms have been investigated so far in the technical literature. The protocol would require limited modifications at the hardware level (the static part of the FPGA and the user accelerator) and at the software level (a special service is needed to securely communicate with the FPGA).

In addition to introducing an extended architecture and related protocols/mechanisms, this work also aims at sound approaches for the formal verification of the introduced solutions. In general, formal verification is a powerful tool that can be applied at any stage of a project life cycle, including early stages as in our case. Applied Pi Calculus has been identified as the formalism of choice. It is a descendant of the Pi Calculus, a process calculus that can express communication of parallel processes, but in addition it introduces constructs that allow the symbolic description of cryptographic primitives and hence it is very suitable for our ultimate goals. The verification phase is automated by a tool, ProVerif, that takes as input a model expressed in Applied Pi Calculus (a

variant of it) with a series of queries and annotations that define security properties on the model, such as authentication or secrecy, and it is capable of proving or denying them. The results of the analysis are encouraging since all the properties defined in this work hold even with a Dolev Yao attacker model.

The rest of the paper is organized as follows. Section 2 presents a general model of a baseline trusted computing platform augmented with reconfigurable hardware, which could potentially be built using currently available technologies. Section 3 introduces a proposal for architecture/runtime-level support as well as a protocol for trusted user applications including an FPGA component. Section 4 addresses key security-related properties and their formal verification. Section 5 concludes the paper with some final remarks.

2 Reference baseline architecture

In this section we define a general baseline architecture model and we identify the key roles and the trusted computing mechanisms that are assumed to be available in the software part of the system. We assume the physical platform to be composed of two main subsystems, a CPU and an FPGA that can be reconfigured at runtime. An instance of this model may be represented by the new Intel multi-chip package solutions embodying a Xeon processors and and Altera FPGA.

CPU. We suppose to have a processor that supports at least the following trusted computing features:

- **Memory isolation**: it is possible to instantiate isolated *containers* in the context of a user application that are physically isolated, from the memory point of view, from the rest of the system. The isolation is enforced at the microcode or the hardware level, so that we can suppose that the privileged software (OS, hypervisor or firmware) has no way to break the isolation mechanism and is, therefore, excluded from the Trusted Computing Base (TCB). In this paper we call the isolated containers instantiated by a user application **User Containers** (UC). We can assume that the isolation mechanism enforced by the trusted hardware protects the restricted memory both from software accesses (intervening in the process of virtual address translation, for instance) and from peripherals that try to access the restricted memory using physical addresses. This kind of memory isolation granularity can be obtained, for instance, by means of SGX mechanisms.

- We also suppose that the each container is accompanied by a **control structure** that stores identifying information about the container such as:

  1. The container **measurement**, that represents a hash of all the container code and data
  2. A **digital certificate** that is signed by the the container's author.

  The measurement is crucial to guarantee the container integrity when it is loaded in memory, while the certificate is useful to establish a relation between containers by the same author (we will see below why this is important). The control structure associated with a container is populated by the trusted hardware runtime whenever the container is loaded and stored in protected memory. It must be only accessible to the trusted hardware, i.e. not even the container associated with the control structure should access it.

- **Key Derivation Function**: It is a primitive that can generate cryptographic keys tied to the container identity and to the hardware that generates them. This is an important function that enables a number of crucial features:

  1. Binding the key to the specific hardware instance that generates it is important to guarantee that the encrypted secret will never be decrypted outside the intended platform. This is crucial because once the secret is encrypted and stored in an untrusted storage or unprotected memory, it can be accessed by malicious applications and leaked to another machine. If this machine has access to the same software (container) then it can decrypt the secret because it can obtain the same key. The key can be bound to the hardware including in the CPU a cryptographic secret that can be used as an input to the key derivation function. This secret should be unique for every processor and stored

in a secure, possibly tamper-resistant storage. An example of such a storage can be the CPU e-fuses. We will refer to this secret as the **CPU Secret**.

2. Binding the key to the container identity enables sealing the data, or, in other words, encrypting them with the key that only the container can generate and storing them in an untrusted medium (unprotected RAM or disk).
3. The key should also be bound to the container signed certificate. In this way containers by the same author can securely share secrets. This is useful, for instance, if the two containers represent different versions of the same software.

Last, we also assume that the key derivation function implements a key anti wear out mechanism. This is implemented by adding another input to the key derivation function, the **anti wear-out** (AWO) field, that is defined by the user and should preferably be a random value.

The instruction through which a container can request the creation of a seal key is the following:

$$CREATEKEY(AWO, Policy, KeyType)$$

where:

- **AWO** is the anti wear out value. It should be random when the key requested is a seal key. A default value can be used when, for instance, the key is a report key that must be generated by two different containers.
- **policy** indicates whether the key should be bound to the container's measurement or to the author's signature. It can assume the values: "Measurement" or "Signature".
- **KeyType** is used to differentiate different type of keys. Other than seal keys the container can in fact request keys for the verification of report (see below).

- **Local attestation scheme**. It is needed in order to let containers authenticate each other through the use of **reports** generated by the trusted hardware. The local attestation process could be articulated in the following phases:

1. The source container requests the report creation to the trusted hardware, specifying the identity (measurement) of the target container that has to authenticate the source container and, optionally, a payload carrying user defined information.
2. The hardware generates the report including the identity information of the source container and the optional payload. Then it computes a MAC tag on these fields to protect the report integrity. The MAC tag is computed using a keyed hash function. The key in input to such a function is generated from the target container measurement provided as input so that it can be the only one to verify that report.
3. The target container receives the report and requests to the hardware the generation of a key bound to its measurement so that it can recompute the MAC tag on the report's fields and compare it to the MAC tag provided along with the report.

The instruction through which a container can request the creation of a report is the following:

$$CREATEREPORT(TargetMeasurement, Payload)$$

Remote attestation is also an important aspect of trusted computing but is not fundamental for the purposes of this work.

FPGA. The FPGA communicates with the CPU through a dedicated bus and accesses the same address space accessed by the CPU. It is composed by two main regions:

- **A dynamically configured acceleration fabric**, which is the region that can be configured at runtime by the software environment and that will be configured with the accelerator provided by the user, or **user accelerator (UA)**. We suppose that the accelerator supports a set of mandatory control/status registers needed to precisely identify the accelerator itself and a set of user defined registers that expose the accelerator functions.

- **A statically configured acceleration infrastructure**, that acts as an interface between the UA and the CPU and implements the physical communication protocol. This infrastructure contains a subsystem, called **FPGA Manager (FM)**, that is responsible for the reconfiguration of the UA.

The communication between the UA and the acceleration infrastructure is implemented by a simple interface that abstracts away the physical channel and implements a load/store semantics in order to enable the UA to send requests and receive responses to/from the main memory. In addition, we also suppose to have a software framework that abstracts the FPGA and provides a set of APIs for the user application to interact with the accelerator. Notice that most of the above trusted computing mechanisms are patterned after SGX and could be directly mapped to an SGX-enabled platform.

3 Extended architecture and trusted computing mechanisms

Under the assumption of an architecture that provides hardware acceleration capabilities paired with CPU support for trusted computing, we will now reason on how to provide a secure interaction between software applications and accelerators, or, in other words, how to extend the trusted computing support to the FPGA fabric. To this aim, we will first consider the extension of the Trusted Computing Base (TCB) to include (part of) the FPGA. Then, we will focus on a mechanism to enable attestation between an FPGA entity and a software container so that they can share a secret key in order, for example, to decrypt a partial bitstream or encrypt sensitive data.

3.1 Threat Model

The interaction between the FPGA and the CPU is not assumed to be secure. An attacker can spoof the communication between CPU and FPGA, e.g. if he/she is able to de-package a multi-chip package CPU+FPGA platform. Since we consider these platforms deployed in an untrusted physical environment we do assume that an attacker has such an ability and, therefore, this assumption leads to defining a passive attacker that can read all the messages exchanged over the bus that links the CPU and FPGA.

Moreover we consider the software environment hostile as well, meaning that all the system software (operating system and hypervisor) is considered malicious. For instance, this allows the presence of corrupted drivers. Such drivers orchestrate the interaction between applications and the accelerator and all the configuration messages, as well as the data messages, are mediated by a driver instance. In other words, a malicious driver or container can potentially modify or replay messages, it could even try to impersonate a legitimate accelerator or software container. In this case the attacker is said to be active, referring to the ability to manipulate the messages on the channel under control.

Secrecy. Secrecy is required in two cases:

- An application that reconfigures a UA with a custom design typically requires the custom design to remain confidential and be stored in memory in encrypted form. Given the above threat model, however, decrypting the bitstream **before** sending it to the FPGA is not enough as the CPU/FPGA channel is not confidential. For this reason, the FPGA should be responsible for the bitstream decryption. To this aim, the user application must communicate the bitstream decryption key to the FPGA. This exchange must guarantee confidentiality of the key.
- An application that uses an accelerator to perform operations on sensitive data wants to exchange such data confidentially and, hence, a mechanism should be provided to guarantee confidentiality in this case.

Authentication. User applications do not necessarily provide their custom accelerators. It is rather likely that they will use an accelerator developed by a third party. This introduces the problem of authenticating the third-party component as an application should share sensitive data only with trusted hardware accelerators.

3.2 Extended architecture-level support

Trusted Computing Base. The first aspect to consider is that the FPGA needs to be included in the Trusted Computing Base alongside the CPU hardware. To achieve this, we need that the CPU and the FPGA share the secrets on which the CPU trusted computing features rely. In particular, if the FPGA had access to the *CPU secret* (Section 2) it could use it as input material for the same Key Derivation Function used by the CPU so that the same generation of keys and reports is possible on the FPGA.

Another crucial aspect of trusted computing solutions is the ability to isolate a piece of software from the external environment. In a software environment this basically means to protect the software memory, which is indeed what our CPU does. Similarly, a UA should be physically isolated from other UAs on the same FPGA. Since the concurrent instantiation of different UAs is not supported by currently available technologies, like Intel CPU+FPGA MCP solutions, we neglected this aspect in this work.

UA Partial Reconfiguration and Measurement We need a way to assure the integrity of the reconfigured bitstream. We will see in the next sections how integrity can be proved to an application that wants to use the UA, but from a hardware point of view we need at least a trusted hash function that can compute a hash checksum on the partial bitstream being programmed. A possible solution is illustrated in Figure 1. A hash block that receives, in parallel with the Partial Reconfiguration Controller, the decrypted bitstream. It can be assumed that the UA exposes an interface that provides its expected measurement so that only if it is equal to the one computed, the *freeze* signal is de-asserted and the UA can start operations.

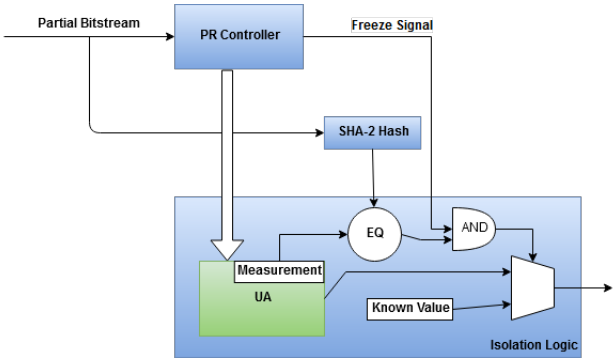

Figure 1: Support to accelerator measurement

3.3 Details of the extended trusted computing mechanisms

In this scenario we consider an application with security requirements that wants to reconfigure an accelerator. We can assume that the bitstream that the application will supply to the FPGA be encrypted, and so the main problem is to give the FPGA Manager (FM), in charge of the reconfiguration, the key for the decryption of the bitstream. To this aim, we can exploit the key derivation features that allows containers by the same author to share a cryptographic key. A problem here is that the user container, accessing the FPGA, and the FPGA Manager, acting as the container that receives the secret, are not from the same author. We can therefore introduce a new container, signed and delivered by the platform manufacturer, that can offer a brokerage service to user containers needing hardware accelerator. Let us call this container **FPGA Interface Container** (FIC). This container does have the same author as the FM, the platform manufacturer, so if we consider the FM equipped with a certificate similar to that of a container (the Container Control Structure), we can securely share a secret between these two entities. Moreover, there is another key advantage in having such an architecture: the FIC can impose an access control policy based on the container identity whenever a container requests the services offered by the FIC.

Once the UA will be configured, we should make sure that it is the expected accelerator, so it should be attested. An container is usually remotely attested by the final user or its creator. Anyway we could avoid to perform a remote attestation for the UA if we consider that there already exists a trusted component executing on the platform, the user container that requested the FPGA reconfiguration. We then rely on such a container to attest the UA and, in case of success, it can provide an activation token to the accelerator in order to activate the part of the IP Core dedicated to the actual function it has to provide.

A first overview of the protocol is the following. We assume that there exists a secure communication channel between the user container and the FIC. This can be established with one of the several mechanisms provided by many user space libraries. The container, hence, can securely

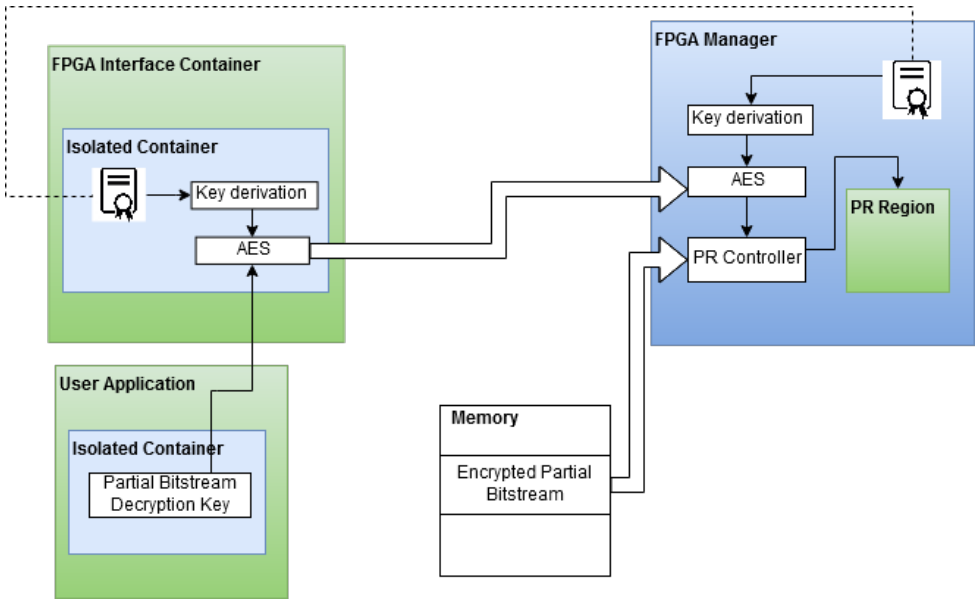

Figure 2: Partial Bitstream decryption key exchange

communicate the partial bitstream decryption Key (PBK) to the FIC and give to it a reference to the encrypted bitstream. At this point the FIC can apply an access control policy if needed and, in case the requesting container is authorized, it can proceed by encrypting the PBK with a key based on the CPU Secret and on its certificate's Signature and sending it to the FM. As we mentioned before, the FM has access to the CPU secret and is signed by the same private key as the FIC and can, therefore, generate the same symmetric key to decrypt the message.

At this point the FM can reconfigure the Partial Reconfiguration (PR) Region. A key function that the FM has to carry out during the reconfiguration is to measure the bitstream being configured. This provides a way for the user container to verify the integrity of the UA. In particular the user container can check that the UA is not altered by these steps: If the FM measures the bitstream as it is being configured we have the equivalent of the container's measurement for the UA. We can then assume that a UA comes with a certificate signed with its author private key, that the FM can request after the configuration. So, as a confirmation of the configuration, the FM can build a report, indicating the identity of the UA, send it to the user container that can attest the UA's identity, and check that it was configured by trusted hardware since the keyed hash in the report can only be generated by that specific hardware.

Keys wear out

The second aspect to consider is key wear out. In fact, since the FIC and the FM will always communicate using the same cryptographic key (the one that the hardware generates from their identities and the CPU secret), this key is subject to wear out, or, in other terms, an attacker could repeatedly give the FIC a plaintext and observe the cyphertext in order to infer the key (which is always the same). To mitigate this problem we can make FIC and FM agree on a common value for the anti wear out field of the key generation function (Section 2) for every new configuration request. This, anyway, involves an additional phase in which FIC and FM agree on an anti wear out (**AWO**) value. In this phase the FIC should choose a random value for AWO and send it in encrypted form to the FM. This is the simplest solution, but actually does not solve the problem because the first message will always be encrypted with the same key. Another way could be to store, both on the processor and on the FPGA, an array of previously agreed AWOs and send, in the first message, only the randomly chosen index of such array. Unfortunately, however, AWO values are typically 256 bit wide, and storing a large array of such values could be infeasible. As an intermediate solution, we could use a relatively small array of pre-agreed AWO values. For the sake of simplicity let us assume that both the FIC and FM have this array stored in their respective address space, encrypted with a seal key known only by them. One of the AWO contained in the array is chosen to create the key that will encrypt the randomly generated AWO which, in turn,

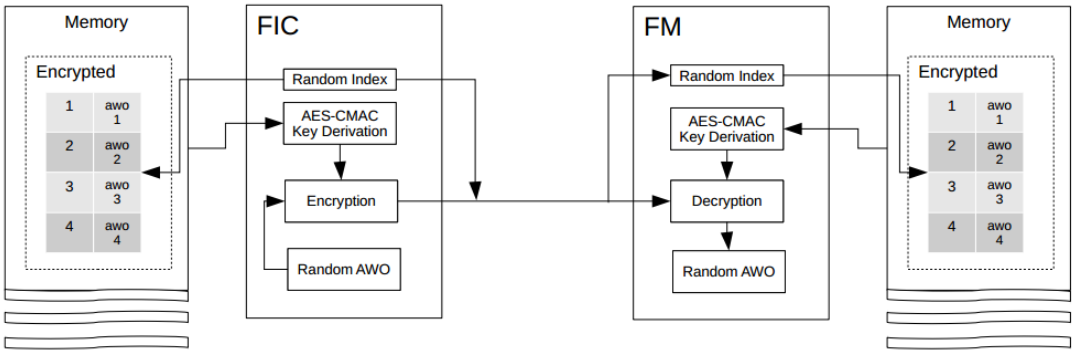

Figure 3: Anti Wear Out Agreement

will be used for the current the session.

There is yet another advantage in using this scheme: the exchanged AWO can act as session identifier in order to avoid replay attacks.

The Protocol

Based on the previous considerations we can then define the protocol for the configuration of a UA by an isolated container. Two phases can be distinguished:

1. The partial bitstream decryption key exchange, in which the user container communicates the decryption key to the FM by using the FIC as a proxy. This phase is, in turn, composed of two sub-phases:
   - The agreement of the AWO between the FIC and FM.
   - The actual communication of the decryption key.
2. The newly configured accelerator attestation, in which the user container verifies, with the support of the trusted hardware, that the UA configured is the intended one.

We will now give the details of the two phases.

Phase 1

In this phase we assume the communication between FIC and UC to be secure. The establishment of a secure communication channel can be accomplished through one of the several mechanisms provided by secure crypto libraries. The steps are the following:

1. The UC initiates the protocol session generating a report through the CREATEREPORT instruction. The report is intended, of course, for the FIC. In the data field of the report structure, the UC writes both the decryption key and a reference to the encrypted bitstream. The UC sends the report to the FIC.
2. The FIC verifies the report and possibly enforces a required access policy. If the container is approved, the FIC generates a random integer i such that $i \in [0, AWOVectorDim)$, where $AWOVectorDim$ is the size of the array containing the pre-agreed AWO values. Hence, it proceeds by fetching $AWOVector[i]$ and calling

   $$CREATEKEY(AWOVector[i], "Signature", FpgaConfiguratioKey)$$

   where the second parameter is the key derivation policy and the third the key name for the FPGA reconfiguration operation. Let us denote the obtained key with $KEY_{vec(i)}$.
3. The FIC generates another random value, this time a random sequence of bits, that represents the AWO for the session: $SessionAWO$.
4. The FIC constructs the message $(i, \{SessionAWO\}_{KEY_{vec(i)}})$ and generates a report through

$$CREATEREPORT(FM, (i, \{SessionAWO\}_{KEY_{vec(i)}}))$$

The first parameter is the identity of the target of the report and, hence, the FM in this case. The report is sent to the FM.

5. The FM verifies the report, so that it can be sure of the integrity of the message. Then, using i, it fetches $AWOVector[i]$ and decrypts the body of the report. In order to prove its identity to the FIC, it generates the message:

$$\{f(SessionAWO)\}_{SessionKey},$$

where $f()$ denotes any deterministic function, even as simple as a subtraction, and $SessionKey$ is the key obtained from $SessionAWO$.

6. The FIC can now authenticate the FM if the decryption succeeds and the decrypted value equals $f(SessionAWO)$, otherwise the session is aborted.

7. Last, the FIC sends the partial bitstream decryption key, encrypted with the session key. An ACK is sent to the UC to indicate that the reconfiguration is taking place.

In this phase the actual exchange of the key happens. Moreover the FIC attests the container identity. Finally the FIC authenticates the FM. This step is necessary because the channel between the two is not secure, hence the FM can not just send a report as this report could be intercepted and then replayed. For this reason, the protocol implements a challenge/response scheme. The challenge is the encrypted session AWO that only the FM can decrypt. The FIC checks that it is talking with the FM only if the latter is able to decrypt the session AWO and send back $f(SessioAWO)$. Still, the FM response could be replayed and used in a different session, but this could not fool the FIC as the replayed $SessionAWO$ would not be the one of the current session.

Phase 2

At this point the UC is reconfigured. Before it can begin to operate, however, it has to be attested so that the user container can check that the reconfiguration was successfully performed and the UC can be activated. Note that the activation token should be kept secret. To do so, the following steps are executed.

1. The FM compares the measurement of the bitstream it has computed with the measurement associated with the user accelerator and if they match it enables the UA to proceed.

2. The UA generates a report through a dedicated interface with the FM. Such report includes the UC's Measurement and its Certificate's signature, just like a container report. The report is then sent to the user container.

3. The user container verifies the report, and if the verification is successful, it provides the UA with an activation token that enables its functional region. There are two ways of providing such a token to the UA, depending on whether the reconfiguring container is from the same author of the UC or not. In the former case the container can directly encrypt the token with its seal key. The UC can generate the same key and decrypt the token. In the latter case, the container can, again, take advantage of the FIC reusing the session key to exchange the token with the FM that, in turn, will give it to the UA. We will consider the latter scenario as it is the most general. It also covers the case of a container that wants to share a secret key with the UA to establish a secure communication channel.

To summarize, we have the following exchanges:

Phase 1

$Message1$	UC $\rightarrow$ FIC :	$R(PBK, UCid)$
$Message2$	FIC $\rightarrow$ FM :	$R((i, \{SessionAWO\}_{Key(i)}), FICid)$
$Message3$	FM $\rightarrow$ FIC :	$\{fn(SessionAWO)\}_{SessionKey}$
$Message4$	FIC $\rightarrow$ FM :	$\{PBK\}_{SessionKey}$
$Message5$	FIC $\rightarrow$ UC :	ACK

Phase 2

$Message6$	UA $\rightarrow$ UC :	$R(NIL, UAid)$
$Message7$	UC $\rightarrow$ FIC :	$ActivationToken$
$Message8$	FIC $\rightarrow$ FM :	$\{ActivationToken\}_{SessionKey}$
$Message9$	FM $\rightarrow$ UA :	$ActivationToken$

where

- R() denotes the CREATEREPORT function in which the first parameter is the data field (NIL stands for an empty data field and the report is only used to attest identity), the second the identity of the sender.
- PBK is the Partial Bitstream decryption Key.

Accessing the trusted user accelerators

We now consider the situation where a user application including trusted software containers needs to use a hardware accelerator. In this case, the key requirement is to ensure authentication between the application and the selected accelerator. Based on the previous setting, we just need here that the two entities exchange reports. For secrecy, this should not be considered mandatory because of the overhead that it can introduce. Consider, for example, the case in which an application wants to use a UA that offers a hardware accelerated symmetric encryption service. It would not make any sense to encrypt the data, send them to the UA that in turn will decrypt the data and re-encrypt them. So, we can imagine that an application may consider tolerating the risk that the information sent to the FPGA will be spoofed and decide not to enforce secrecy.

On the other hand, if secrecy is required, then we need to exchange a key. This key will be generated by the user container because, in any case, sealed data must be bound to its identity. This case is very similar to the previous scenario, particularly Phase 2, where the UC must attest the UA, and, hence, the protocol defined above can be reused as-is. The only difference is that after the FM receives the UC/UA key, it must pass it to the UA, which is hence supposed to provide an interface to save the key. Moreover, once the key has been delivered to the UA, in addition to attesting the UA, the container could require a proof that the UA actually has the intended key. This can be accomplished by letting the user container generate a nonce that will be sent to the UA along with the encryption key, encrypted by the UA, and sent back.

3.4 Outline of implementation-related aspects

In this section we will consider how the solution outlined above could be implemented in practice.

FPGA side. For what concerns the FPGA side, we should consider the following implementation aspects:

- The first issue is the storage of the CPU secret. Modern FPGAs provide two ways for securely storing keys on chip [1]: the **volatile** storage is composed of reprogrammable and erasable battery-backed RAM registers, while the **non volatile** storage consists of fuses-based registers that are one time programmable. Both methods are implemented such that the extraction of the key is difficult. However, if we consider that the static part of the FPGA containing the FM is encrypted, then necessarily one of the two secure storages must be used to preserve the decryption key for that static FPGA part.
- Then key derivation and encryption and decryption modules have to be implemented. The key derivation process could implement the method described in FIPS SP 800-108, with AES-CMAC as pseudo random function, while AES can be used for encryption. Both the AES module and the key derivation function should be included in the FPGA static part that contains the FM, and both the UA and the FM should have the possibility to access these modules.
- Similar considerations hold for the generation and verification of reports. The main functionality of report creation and verification is represented by the keyed hash function that computes a MAC tag on the report's body. This function could be implemented with a block cypher-based MAC (CMAC) based, in turn, on 128-bit AES.

- Additional logic should be introduced in the static part of the FPGA in order to implement the protocol. This should include a control unit to orchestrate the exchange of messages and the hardware to perform the measurement of the bitstream.
- Last, as for the accelerator, it should include additional mandatory registers that represent its measurement and its signature so that the FM can verify such values.

Software Side. The privileged container introduced (FIC) is a trusted piece of software that allows secure reconfiguration of the FPGA, beside providing a way to exchange secrets between the application and the UA. User applications allegedly interact with the FPGA through a software framework, so such a service should be integrated within the architecture. That means that new APIs should be added implementing the secure reconfiguration of an accelerator or the secure ownership acquisition. The new functionality backing the secure communication should, of course, be implemented by structuring the application in containers, in order to make the service trustworthy.

4 Formal verification

A major goal of this work was to prove the soundness of the proposed solution. We do not have an implementation since no physical platform was available for demonstrating the proposed architecture-level support. Nevertheless, our formal verification addresses the high-level specification of the protocol and does not depend on a specific implementation. Many formalism and techniques exist that aim at the verification of security protocols like [17, 18, 21]. For an extensive survey the reader can refer to [22] and [5].

The choice that most suits the needs of this work is the **Applied Pi Calculus** [2]. It is explicitly designed for describing and analysing security protocols and, in fact, it extends the Pi Calculus [20] by including primitives for expressing encryption and decryption and assumes a Dolev and Yao attacker model [8], where the attacker has full control over the communication channels. Moreover, in this formalism, the intruder is not even modelled explicitly, but it is implicitly represented as another process of the calculus. The formalism also allows modelling a system as a group of processes interacting with each other through the exchange of messages. Applied Pi Calculus (like its parent, Pi Calculus) also supports the concept of channel modelling the means by which information flows. Through the definition of a set of function symbols (**signature (Σ)**) and their relations (**equational theory**), described as equations, the Applied Pi Calculus enables the expression of the rich semantic of a protocol. This, along with the fact that all the operations of a participant to the protocol can be described in terms of processes, makes it a very powerful methodology because the analysis takes into account all the internal operations carried out by protocol principals and not only the messages they exchange. Moreover this representation is closer to a possible implementation unlike, for instance, a BAN Logic model which is more formal but more difficult to relate to a possible protocol implementation. On the other hand, since the cryptographic primitives are defined through a set of equations on function names, they are only symbolic, meaning that they are seen as black box operations that do not expose the internal behaviour. In other words, the underlying assumption is that cryptography is perfect. Hence models expressed in Applied Pi Calculus do not take into consideration attacks on the cryptographic primitives. The reader may refers to [2, 24] for details on Applied Pi Calculus.

Another advantage of the chosen formalism consists in the availability of a number of automated verification tools. The most prominent of such tools is probably **ProVerif** [6]. It is a symbolic protocol verifier that internally represents the protocol by **Horn clauses**. Horn clauses are first order logical formulas in the form $F_1 \wedge \cdots \wedge F_n \Rightarrow F$ where F_1, . . ., F_n, F are *facts*. This representation keeps relation information on messages but introduces abstractions that do not guarantee termination. The input of ProVerif is a model expressed in a variant of the Applied Pi Calculus. This language, called typed Pi Calculus, supports several cryptographic primitives expressed as rewrite rules or equations. Another input of ProVerif is a set of properties to be tested on the modelled protocol. The tool is able to prove secrecy, authentication, and some observational equivalence properties. Such properties are expressed as queries that are translated into derivability queries on the Horn clauses to which the protocol is mapped. The resolution process of ProVerif tries to determine whether a fact is derivable from the clauses. If the fact is not derivable, then the property is proved. On the contrary, if the fact is derivable an attack may be found. Note that due to the abstraction introduced by Horn clauses, there is the possibility of false attacks. If the attack

is true, however, ProVerif is able to reconstruct the attack in terms of messages exchanged. If it cannot reconstruct the attack then the output is "I don't know", meaning that the found derivation is probably a false attack. Secrecy and authentication are the most important security properties for most protocols and ProVerif provides a simple way to query such properties.

Secrecy of a term (secret) can be easily queried through the following command:

```
query attacker (secret).
```

Authentication is based on correspondence properties and, referring for instance to a client/server scenario, it can be queried in ProVerif through:

```
query event (serverEnd) ==> event (clientStart)
```

meaning that if serverEnd occurs (the server has accepted the client) then clientStart has occurred before (the client has indeed sent a request to the server), or in other words, it cannot happen that the server accepts a request that is not from the client.

4.1 The Applied Pi Calculus instance used in this work

In this section we give a description of the instance of the Applied Pi Calculus used to model the protocol of Section 3. The first step is to define function symbols and an equational theory for modelling the messages and the operations performed by the protocol participants.

4.2 Σ and Equational Theory

For this study, Σ was chosen to have the following grammar for terms:

$M, I, U, V, P, A ::=$	Terms
c, n, s	names
K, T, k, r, m, z, a	variables
$senc(U, I)$	symmetric encryption
$sdec(U, I)$	symmetric decryption
$identity(I)$	generate container identity from initialization token
$H(U, I)$	keyed cryptographic hash function
$derMaterial(V, P, A)$	derivation material to feed key generation function
e, g, d	constants for derivation policy and default anti wear out
$sealKey(I, M)$	symmetric seal key generation function
$repKey(I, V)$	key used to compute the MAC of a report
$repVerKey(I)$	key used to verify a report
$report(U, V)$	constructor for report
$F_U^{report}, F_V^{report}$	projections for report
$fn(V)$	generic deterministic function
$actToken(U)$	Generate an activation token bound to a secret

Shared key cryptography. In order to model shared key cryptography two binary symbols were introduced: $senc(\_, \_)$ and $sdec(\_, \_)$ for encryption and decryption, respectively. The equation that rules their use is:

$$sdec(senc(x, k), k) = x$$

where x represents the message to be encrypted and k the symmetric key.

Container Identity. The container identity is bound to an initialization token that proves its identity. A function was introduced that, given an initialization token, returns the container identity. Of course there is no inverse function that can return a token from the container identity as the token must remain secret while the identity of the container is public.

Keyed cryptographic hash. This function is used to calculate a keyed hash value on reports. It is modelled with a binary function, $H(\_,\_)$, that has no equations in order to express the one-way nature of hash functions. Note that $H(x,k) = H(x',k') \iff x = x'$ and $k = k'$ which implies that this hash function is collision free.

Derivation material. The seal key generation process is fed by a series of inputs collectively named *key derivation material*. The $derMaterial(\_,\_,\_)$ symbol allows generating derivation material from the requester identity, the policy for key derivation, and a value used as anti wear out. The latter value plays a key role in the protocol.

Seal key generation function. This constructor models the key generation process that creates symmetric seal keys from a token owned by every initialized container and from a set of values, the key derivation material, including the container's measurement and an anti wear out value. It is analogous to the previous defined hash function as it is not invertible and, for the same derivation material, generates the same keys.

Report key generation functions. The $repKey(I,V)$ constructor generates the keys that are used in the keyed cryptographic hash function to hash reports. Its inputs are the initialization token of the container requesting the key (only an initialized containers can request keys) and the identity of the target container. On the other hand, the $repVerKey(I)$ constructor generates a key used to verify a report. The two constructors are linked by the equation:

$$repKey(y, identity(x)) = repVerKey(x)$$

where y is the initialization token of the container generating the report, x is the initialization token of the target container, and $identity(x)$ is the identity of the target container. This equation ensures that only the target container can verify the report as if the identity specified in $reportKey$ does not correspond to token x, the generated keys will not match.

Reports. The *report* function symbol introduces in Σ the creation of reports. A report is built using $report(x,t)$, where x is the data to be inserted in the body of the report, and t is the initialization token of the container generating the report. In order to extract such information from the report, two inverse un-ary functions are added, one to extract data and one to extract the identity of the creator from the report. The equations that rule such behavior are:

$$F^{report}_{data}(report(x,t)) = x$$
$$F^{report}_{identity}(report(x,t)) = identity(t)$$

where *identity* is the constructor defined above. This solution reflects the hypothetical report mechanism because, since the initialization token is never exported outside the scope of the software container (or accelerator) process, only the container in possession of the token can generate reports that attest the container's identity. Hence the attacker cannot forge reports carrying another container's identity.

Generic function. This symbol just models a generic deterministic operation on its input. It is used in the challenge/handshake exchange to verify that the receiver has been able to decrypt the message. To prove the decryption, the receiver calculates $f(x)$ where x is the challenge and sends this value to the initiator. The initiator checks it to accept the receiver. The comparison is based on the equation:

$$fInv(f(x)) = x$$

where $fInv$ is the inverse function of f

Activation token. Last, the $actToken$ symbol allows generating activation tokens in Σ. A container uses this function to generate a token that activates the function specific region of an accelerator. We let this token be bounded to the key that decrypts the bitstream for the UA.

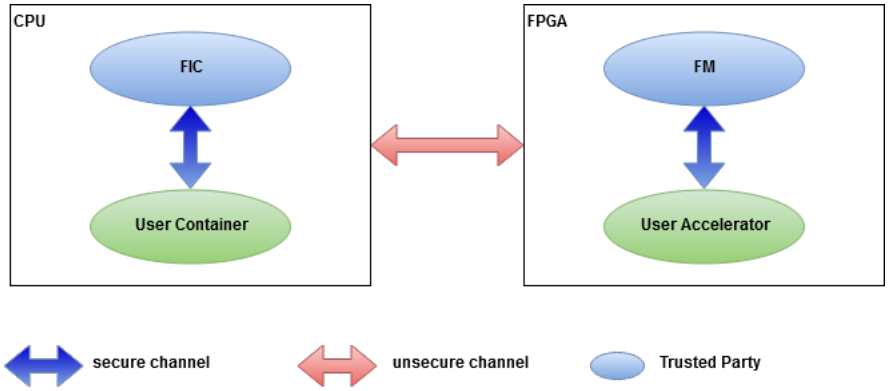

Figure 4: Processes and channels

4.3 The Applied Pi Calculus model

In this section we will discuss the representation of the communication channels, the identified roles, and the attacker in the Applied Pi Calculus model.

Communication

A first exchange of messages takes place between the UC and the FIC. As specified before, we assume that these two containers have already established a secure communication channel. This is easily achievable by making the two containers attest each other, in order to establish their identity, and then perform a Diffie-Hellman exchange to establish a shared key to encrypt their communications. As a consequence, we can model this channel as a private Pi Calculus channel. The attacker cannot interfere with this channel. This reflects our assumption that CPU mechanisms are considered secure and trustworthy.

The relation between FM and UA is similar to that of UC and FIC, so the communication between the former principals is also considered secure. This is a reasonable assumption as this communication takes place on the FPGA and it would be extremely expensive for an attacker to tap the communication within the chip. Moreover we assume that a malicious UA is not able to obtain any information on the communication in place between UA and FM. Finally, we have the communication channel between the FPGA and the processor. We assume that an attacker can successfully read all communication on the communication bus between them and hence we modelled it as a free Pi Calculus channel.

Roles and processes

As explained in Section 3, there are four different roles that participate in a protocol run: the User Container, the FPGA Interface Container, the FPGA Manager, and the User Accelerator. In our model we refer to every principal by their identities using the variables ID_A, ID_B, and so on. The context:

$$IDENTITIES^A[\_] = \nu T^A.(ID_A = identity(T^A) \mid [\_])$$

is associated to every principal A, so that the use of the initialization token T^A is restricted to the process in the context. Moreover, the identity ID^A is exported without revealing the token T^A to the environment since no inverse function exists for $identity(\_)$.

Four processes were defined that implement the roles defined above. Each of them follows the steps defined informally in Section 3. The FIC and FM processes correspond to the homonymous architectural components and implement the core of the protocol. The FIC accepts requests from a UC and cooperates with the FM to securely reconfigure a partition. The UC and UA processes represent the users of the protocol and must conform to its specification. The UC can request the configuration of a partition and, once this is done, it can attest the UA and then provide it with an activation token that will be securely transmitted to the UA through the FIC and FM.

Following is the implementation of these processes in Applied Pi Calculus. The communication channels are:

- **CsecC** (Containers secure channel): the secure channel between User Container and FPGA Interface Container.
- **FsecC** (FPGA secure channel): the secure channel between User Accelerator and FPGA Manager.
- **pubC**: the public channel that links CPU and FPGA.

Each process is created with a token T_x. Note that the FIC and the FM are created with the same token. This reflects the fact that these two components are conceptually one trusted entity distributed over two nodes. Moreover every UC process is associated with the identity of the UA it wants to configure (ID_{ua}) and, vice versa, every UA process is associated with the identity of the reconfiguring container (ID_{uc}).

process User Container

$UC^{T_e}_{ID_{ua}} = \quad !\nu K_{PB}.$	the Partial Bitstream decryption key
$let(K_r = repKey(T_e, ID_{fic})\ in$	report key
$let(r_{uc} = report(K_{PB}, identity(T_e))\ in$	report (data = decryption key)
$\overline{CsecC}(r_{uc}, H(r_{uc}, K_r)).$	output report and MAC of report
$CsecC(= identity(T_e), ACK).$	wait for the ACK
$pubC(= identity(T_e), r_{ua}, m_{ua}).$	wait for the UA to send its report
$let(K_v = repVerKey(T_e))\ in$	report verification key
$let((= m_{ua}) = H(r_{ua}, K_v))\ in$	verification of the report
$if(ID_{ua} = F^{report}_{identity}(r_{ua}))\ then$	check if identity is the one expected
$let(t = actToken(K_{PB})).$	activation token
$\overline{CsecC}(t, ID_{ua})$	send activation token through FIC

process FIC

$FIC^{T_i}_{ID_{fm}} = \quad !CsecC(r_{uc}, m_{uc})$	wait for incoming reports
$let(K_v = repVerKey(T_i))\ in$	report verification key
$let((= m_{uc}) = H(r_{uc}, K_v))\ in$	report verification
$\nu a.$	new session anti wear out
$let(z = derMaterial(identity(T_i), g, d))\ in$	derivation material (default anti wear out)
$let(K_d = sealKey(T_i, z))\ in$	seal key
$let(K_r = repKey(T_i, ID_{FM}))\ in$	report key
$let(r_{fic} = report(senc(a, K_d), identity(T_i)))\ in$	report (data = encrypted session anti wear out)
$\overline{pubC}(r_{fic}, H(r_{fic}), K_r)$	send report to FM
$pubC(x)$	wait for FM response
$let(z' = derMaterial(identity(T_i), g, a))\ in$	derivation material (session anti wear out)
$let(K_{session} = sealKey(T_i, z'))\ in$	session seal key
$let(= fn(a)) = sdec(x, K_{session})\ in$	check the challenge response from FM
$\overline{pubC}(senc(F^{report}_{data}(r_{uc}), K_{session})).$	send PB decryption key to FM
$\overline{CsecC}(F^{report}_{identity}(r_{uc}), ACK).$	send ACK to user container
$CsecC(y, ID_{ua}).$	wait for the activation token
$\overline{pubC}(senc(y, K_{session}), ID_{ua})$	send to FM the encrypted activation token

process FM

$FM^{T_i}_{ID_{fic}} = \quad !pubC(r_{fic}, m_{fic})$	wait for incoming reports
$let(K_v = repVerKey(T_i))\ in$	report verification key
$let((= m_{fic}) = H(r_{fic}, K_v)))\ in$	report verification
$let(z = derMaterial(identity(T_i), g, d))\ in$	derivation material (default anti wear out)
$let(K_d = sealKey(T_i, z))\ in$	seal key
$letx = sdec(F^{report}_{data}(r_{fic}), K_d)$	decrypt session anti wear out
$let(x' = fn(x))$	apply fn to x
$let(z' = derMaterial(identity(T_i), g, x))\ in$	derivation material (session anti wear out)

$let(K_{session} = sealKey(T_i, z'))\ in$	session seal key
$pubC(senc(x', K_{session}))$.	send $fn(x)$ to FIC
$pubC(k)$.	wait for FIC to send PBK
$let(k' = sdec(k, K_{session}))\ in$	now the FM can reconfigure the UA
$\underline{pubC}(y, ID_{ua})$.	wait for the activation token
$\overline{FsecC}(ID_{ua}, sdec(y, K_{session}))$.	send the activation token to the intended UA

process UA

$UA^{T_f}_{ID_{uc}} = \quad !\nu K_{PB}.$	
$let(K_r = repKey(T_f, ID_{uc}))\ in$	report key
$\underline{let}(r_{ua} = report(NIL, identity(T_f)))\ in$	report (data = NIL)
$\overline{pubC}(ID_{uc}, r_{ua}, H(r_{ua}, K_r))$	report (send report to UC)
$FsecC(= ID_{uc}, a)$	report (wait for the activation token)

This model abstracts away some implementation details. In particular, we do not model the hardware initialization check involving the measurement of the UA bitstream. This does not hurt the validity of the model, as a failed initialization would imply that the accelerator would never send its report to the UC and, hence, the session would be aborted.

4.4 Verification with ProVerif

We will now present the translation of security properties, to be tested in ProVerif, in events and queries. Then, we will carry out a series of tests adjusting the resolution strategy of ProVerif and introducing a malicious process to simulate key compromise.

Security properties

The proof for security properties is carried out in ProVerif using correspondences. Correspondences express a property that we would like to hold for the model provided. We will now describe the correspondences defined.

1. Authentication of User Container to FIC. A user container is granted access to the FPGA for reconfiguration only if the FIC can attest its identity. Actually this step could be skipped as the UC and FIC already share a secure channel and this should imply that they already have authenticated each other. Anyway, we still define the following correspondence:

event (FICAuthenticateUC(MAC, UCIdentity, Key)) ==>
event (UCstartAuthToFIC(MAC, UCIdentity, Key)).

where *MAC* is the MAC tag computed on the report that the UC creates, *UCIdentity* is the signature of the container's certificate, *Key* is the decryption key for the partial bitstream, included in the report. This correspondence means that if the FIC has received a report from the container *UCIdentity*, with a MAC equal to *MAC* and *Key* included in the report data field, then a container should have sent it before.

2. Mutual Authentication of FIC and FM. The exchange of messages between FIC and FM starts with a challenge/response. We can relate the challenge with the response by using the correspondences:

event (FIC_Rec_Resp_From_FM(SessionAWO)) ==>
inj-event(FM_Send_Resp_To_FIC(SessionAWO)).
event (FM_Rec_Report_From_FIC(SessionAWO)) ==>
event(FIC_Send_Report_To_FM(SessionAWO)).

The first one tells us that if the FIC receives a challenge response from the FM then the FM must have sent such response with the same *SessionAWO* sent by the FIC. This correspondence is injective because the event on the right must be distinct for every occurrence of the event on the left. The second one expresses the same concept in the opposite direction, but it is not injective as the challenge from the FIC could actually be replayed and the FM would not have a way to know that. This, anyway, is not a concern as the replayed message would not allow the attacker to know the session key in any case.

3. Attestation of UA to UC. Once the UA process is started, it sends a report to the UC in order to attest its identity. We prove that the attestation process in not manipulated by the attacker with the correspondence:

event (UCAttestUA(MAC,UAIdentity, UCIdentity)) ==>
event (UAstartAttestUC(MAC, UAIdentity, UCIdentity)).

In other words, if the user container identified by *UCIdentity* has received a report with the MAC *MAC* from the UA identified by *UAIdentity*, then the event UAstartAttestUC must have been executed by the UA with *UAIdentity* with the same parameters received by the user container.

4. Send and Receive of Activation Token. This correspondence is used to check that when the UA receives an activation token, it is indeed sent by the container that reconfigured it.

event (UAReceiveActivationToken(UAIdentity, UCIdentity, token)) ==>
event (UCSendActivationToken(UAIdentity,UCIdentity,token)).

Secrecy. Last, we want to be sure that the attacker cannot obtain the decryption key. The session anti wear out value agreed between the FIC and FM is a further key value that, if compromised, can break the the secrecy guarantee on the key, so we also query secrecy on this value. Finally, the activation token must also remain secret or otherwise the attacker could activate an UA that does not successfully attest its identity to the reconfiguring container. The queries for secrecy are the following:

```
query attacker (new UCBitStreamKey).
query attacker (new SessionAWO).
query attacker (new ActivationToken).
```

Assumptions. In order to simplify the proof of such properties, we can define assumptions on the attacker's knowledge that make ProVerif exclude some hypotheses and therefore speed up the resolution process. The assumptions made are the following:

```
not attacker (PRIVILEGED_TOKEN).
not attacker (new UCINITTOKEN).
not attacker (new UAINITTOKEN).
```

Basically, we say that the attacker cannot know the container's tokens used to generate the keys. This is a reasonable assumption as the tokens are used to model the fact that the identities used in the key derivation process are automatically and safely chosen by the trusted hardware on the basis of the calling container.

4.5 Tests

A series of tests were carried out for various settings of ProVerif regarding the resolution process. In particular, we considered the following settings:

- **set setIgnoreType**: this setting determines how ProVerif treats types. If it is set to true then ProVerif ignores types, allowing the attacker to send ill-typed terms, while when set to false the protocol complies with the type system and hence ill-typed terms are detected. When this setting is true there are more chances to find an attack because type-flow attacks are detected; on the other hand, when set to false the state space is smaller and the verification faster.
- **set attacker**: we can choose to have an **active** or a **passive** attacker. A passive attacker cannot send messages, but only read and do computation.

The protocol was run with all the four combinations of the above settings. All the defined properties were proved to hold.

Compromised Keys. We cannot use the **keyCompromise** setting, which models compromised keys for some sessions of the protocol, because this setting is incompatible with the use of phases (which our model uses). Consequently, we represent the compromised sessions with a new process that outputs its token on the public channel. First we model the violation of some user container's tokens by introducing the following process:

```
let DishonestContainer
        (DISHONESTCONTSTRUC: controlStructure, DISHONESTTOKEN: initToken) =
                out(pubChannel, (DISHONESTCONTSTRUC, DISHONESTTOKEN)).
```

and then we let the attacker know the token of the FIC and FM with the process:

```
let DishonestContainer
        (DISHONESTCONTSTRUC: controlStructure, DISHONESTTOKEN: initToken) =
                out(pubChannel, PRIVILEGED_TOKEN).
```

As expected, in the first case all the properties still hold. In fact this is the case where the attacker has control of a malicious container. This container, however, cannot do much to interfere with the exchange of the key. On the contrary, when the attacker has the PRIVILEGED_TOKEN, and hence controls the privileged container, all the properties are violated. In particular the attacker can obtain the decryption key. This is not surprising since compromising this token means that the attacker can impersonate the FIC and/or the FM. For this to happen, an attacker needs to circumvent the trusted hardware and make it believe that it is actually a privileged container, nullifying all the assumptions on the CPU security guarantee. To do that it would be necessary to overwrite the attacker container's control structure, which is stored in protected memory and is not even accessible by the container associated with the control structure. Only the processor microcode (or hardware) can access that particular memory region and, therefore, the attacker should be able to compromise the CPU microcode or to physically inject particular values at a specific location of the RAM chip.

5 Conclusions and future work

This work proposed an architecture and a protocol for trusted platforms supporting secure application containers that include an FPGA part. The solution can be mapped to existing trusted computing solutions, e.g., Intel SGX, with limited impact mostly related to the software-side and the FPGA-side runtime support. Importantly, a model of the protocol and the underlying trusted computing mechanisms was developed to formally verify its properties. The formalism adopted for the model was the Applied Pi Calculus, a process calculus specifically designed to verify security protocols. The verification was supported and automated by means of the ProVerif tool. The results were encouraging as the modelled security properties were proved to hold under reasonable assumptions regarding the platform integrity.

As a future work, we plan to investigate the actual technology mapping of the proposed architecture/protocol and deeper implementation-related aspects. These include the possible extension of FPGA software runtime layers, e.g. Intel Open Programmable Acceleration Engine (OPAE), the role of memory protection mechanisms enforced to the FPGA accelerators as well as the security implications of relaxed memory consistency models, particularly when different physical channels are used as in the CCI-P interface provided by Intel Xeon+FPGA platforms.

References

[1] *Intel Arria 10 Core Fabric and General Purpose I/Os Handbook.*

[2] Martín Abadi and Cèdric Fournet. The Applied Pi Calculus: Mobile values, new names, and secure communication. *[Research Report] ArXiv.*, 2001.

[3] Baumann Andrew, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems*, 2015.

[4] Hassan Artail, Mazen A. R. Saghir, Mageda Sharafeddin, Hazem Hajj, Abdulrahman Kaitoua, Raghid Morcel, and Haitham Akkary. Speedy Cloud: Cloud computing with support for hardware acceleration services. *IEEE Transactions on Cloud Computing*, 2017.

[5] Bruno Blanchet. Security protocol verification: Symbolic and computational models. *Proceedings of the First international conference on Principles of Security and Trust*, 2012.

[6] Bruno Blanchet. Modeling and verifying security protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends in Privacy and Secruity*, 2016.

[7] Bruno Blanchet. Symbolic and computational mechanized verification of the ARINC823 avionic protocols. *Diss. Inria Paris*, 2017.

[8] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 1983.

[9] Arnautov Sergei et al. SCONE: Secure linux containers with Intel SGX. *OSDI*, 2016.

[10] Ferraiuolo Andrew et al. Komodo: Using verification to disentangle secure-enclave hardware from software. *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[11] Liang Xueping et al. Man in the cloud (MITC) defender: SGX-based user credential protection for synchronization applications in cloud computing platform. *IEEE 10th International Conference on. IEEE*, 2017.

[12] Pai Suhas et al. Formal verification of OAUTH 2.0 using alloy framework. *Communication Systems and Network Technologies (CSNT)*, 2011.

[13] Schuster Felix et al. VC3: Trustworthy data analytics in the cloud using SGX. *Security and Privacy*, 2015.

[14] Sinha Rohit et al. Moat: Verifying confidentiality of enclave programs. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[15] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet dossier. Technical report, Symantec, 2011.

[16] J.Fowers, G.Brown, P.Cooke, and G. Stitt. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. *SIGDA Int'l Symp. on Field Programmable Gate Arrays, pp. 47-56*, 2012.

[17] Millen J.K., Clark S.C., and Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, 1987.

[18] Burrows M., Abadi M., and Needham R. A logic of authentication. *Proceedings of the Royal Society of London*, 1989.

[19] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

[20] Robin Milner. Communicating and mobile systems: the Pi-Calculus. *Cambridge University Press*, 1999.

[21] Durgin N., Mitchell J.C., and Pavlovic D. A compositional logic for proving security properties of protocols. *Journal of Computer Security*, 2003.

[22] Martin Pitt. Modeling and verification of security protocols. *Dresden University of Technology, Advanced seminar paper*, 2012.

[23] Mark D. Ryan. Cloud computing security: The scientific challenge, and a survey of solutions. *The Journal of Systems and Software*, 2013.

[24] Mark D. Ryan and Ben Smyth. Applied Pi Calculus. *School of Computer Science, University of Birmingham, United Kingdom*, 2010.

[25] Trautman, Lawrence J., Ormerod, and Peter C. Corporate directors' and officers' cybersecurity standard of care: The yahoo data breach. *American University Law Review*, 02/2017.

[26] Y.Shan, B.Wang, J.Yan, Y.Wang, N.Xu, and H. Yang. FPMR: Mapreduce framework on FPGA. a case study of RankBoost acceleration. *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays. ACM*, 2010.