Improving FPGA Security with Merged Hardware Isolation Techniques

ABSTRACT

Field Programmable Gate Arrays (FPGAs) platform security management continues to be a strong area of concern despite recent increased adoption and integration of FPGAs into commercial scale systems, namely cloud computing systems. One of the technical problems in the FPGA security management area has been the lack of hardware primitives to support multi-tenancy while enforcing proper domain isolation. In this tutorial, we present recent progress that has been made to address this issue. Specifically, we present hardware isolation mechanisms that can be used to enable domain separation on FPGA based systems. We demonstrate the viability of these mechanisms through a software-hardware codesign implementing a simple TSL/SSL encryption application.

CCS CONCEPTS

•Security and privacy →Embedded Systems Security; Hardware Security Implementation; •Hardware →Reconfigurable Logic Applications;

KEYWORDS

FPGA Security, Hardware Isolation, IP Containerization, CAPSL

1 INTRODUCTION

Many of today's critical embedded systems are increasingly relying on FPGA-based system-on-chip (FPGA SoCs) because of the useful balance between the performance, scale, flexibility, and rapid time to market they provide. This was recently exemplified with the recent Audi announcement that its 2018 A8 world's first Level 3 autonomous driving system will feature Altera's Cyclone FPGA SoCs for object and map fusion processing tasks [19]. Though, it's not just in embedded systems space where we are seeing accelerated adoption of FPGA platforms, as they are also being continuously integrated into commercial scale cloud computing systems and data centers as evidenced by Amazon's recent announcement of providing cloud computing instances with FPGAs (EC2 F1) [1].

This increased adoption of FPGA into commercial scale cloud computing systems has highlighted the issue of FPGA's lack of hardware primitives, conceptually similar from a security perspective to a CPU's IO memory management unit (IOMMU), to support multi-tenancy while enforcing proper domain separation among the tenants (i.e. multiple applications utilizing the same FPGA) [8], [6], [7]. This lack of native support for isolation creates security concerns where: 1) Since FPGA accelerators tend to run with full hardware access, a single accelerator vulnerabilities can be exploited through traditional system software stacks or if malicious, can bring down a shared compute host [8]. Or 2) tenants are not able to protect confidentiality and integrity of their application data not only from other tenants, but also from other unauthorized parties including privileged system software, such as the hypervisor [2].

An effective solution to the problem of lack of proper domain separation on FPGAs should be able to enable scaling of security across a range of system parameters, such as power and performance, without significant drops in coverage. We believe, and intend to demonstrate in this tutorial, that this can only be achieved through some form of combination of measures that protect against hardware vulnerabilities directly at the hardware level and measures that enforce domain separation at the software level. In this tutorial, we look at a set of research works that individually attempted to address the problem of lack of isolation support on FPGAs. We demonstrate how certain elements of these works can be put together to provide an effective and comprehensive solution to this problem. We demonstrate the viability of this solution through a real-world software-hardware codesign application which implements a simple TSL/SSL encryption.

The remainder of this work is organized as follow: Section 2 starts by describing the specific use case scenarios and security concerns this method addresses. Section 3 gives an overview of research works that form the basis of our comprehensive solution. Section 4 and Section 5 demonstrate how a combination of the discussed approaches can be used to provide proper domain isolation on FPGA platform. Section 6 concludes the paper.

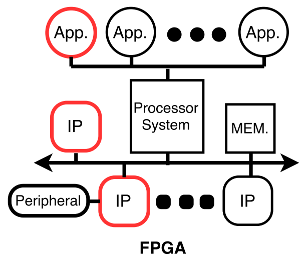

Figure 1: Embedded System Threat Model
The adversary controls the FPGA SoC and can execute malicious code (non-trusted application + accelerators in red) remotely and launch a denial-of-service attack to an IO component.

2 THREAT MODEL

This section describes the threat model for two use case scenarios our approach targets. We first look at the attack surface in FPGA-based embedded systems and then, we look at FPGA security concerns in cloud computing systems integrating FPGAs.

FPGA'18, Monterey, CA, USA
2018.

2.1 Embedded Systems

The illustration of our threat model in Figure 1 shows the first use case scenario our approach targets. We assume an adversary who can introduce hardware trojans in the IPs during development (supply chain attack). We assume this adversary will later have a method of remote access to the FPGA So) which contains malicious IPs. We assume the adversary is aware of the trojans activation mechanisms and can activate them externally by executing some malicious input for example.

In FPGA-based SoCs, we do not currently have hardware-level support mechanisms that would prevent unauthorized bus data access by a bus component. We rely on bus protocols which generally assume that bus components are going to adhere to the protocol and wait for their turn to broadcast. However, this is not always the case since a malicious component can unilaterally decide to access the bus and either eavesdrop or launch a denial-of-service (DoS) attack to the system peripherals.

2.2 FPGAs in the Cloud

The fast adoption of FPGA accelerators in cloud computing and the probable future adoption in personal computers raises additional security concerns. Cloud providers such as Amazon that provide infrastructure as a service, must address security threats posed by malicious IPs residing on FPGAs without affecting the performance of the system. In general, security is enforced in cloud computing using domain separation. Users are assigned virtual instances (VIs) which are then scheduled on physical resources by a hypervisor. The hypervisor controls the access to the hardware and ensures that every virtual instance remains active within its limit. As shown on Figure 2a), such an approach requires resource virtualization, for which no viable solution currently exists in FPGA. As a result, whole FPGAs must be assigned to small accelerators thus wasting resource, or the execution on the FPGA must be emulated in the hypervisor with performance degradation.

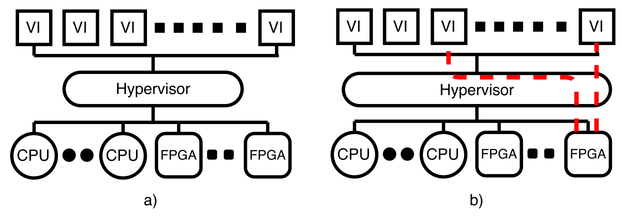

Figure 2: FPGA Provisioning in IAAS-based Cloud Computers

FPGAs are virtualized in (a) and in (b), tenants are granted direct access to the FPGAs.

An optimal use of FPGA resources would allow the programmable logic (PL) fabric to be shared among IPs, sometimes from different tenants for a better FPGA utilization without performance degradation. With several IPs on the same FPGA, there is currently no mechanism to guarantee that some IPs will not perform malicious activities, even in the presence of strong protocols. The classic approach to enable IPs on the FPGA is through registers directly controlled by the hypervisor (for the purpose of domain separation), and accessible to the corresponding instance (Figure 2b). The problem here is that a trojan hidden within an IP can bypass protocols and place data on FPGA lines to negatively affect the system. Since the connections are well known, nothing prevents an IP designer to tap long FPGA lines and read information from neighboring modules or to write on some line randomly to DoS the design on the FPGA hosted in the cloud. As we will show in this tutorial, hardware isolation can address domain separation in hardware to provide an increased protection in security critical architectures.

3 HARDWARE ISOLATION MECHANISMS

Protection of sensitive data in networked and embedded systems has become a critical problem with far-reaching financial and societal implications. Among the ten paradigms (Deception, Separation, Diversity, Consistency, Depth, Discretion, Collection, Correlation, Awareness, Response) used to address cybersecurity, separation is one of the most effective while being relatively easy to implement. Thus separation is increasingly used to address security, in particular in web applications where security concerns have been the highest ([9, 13, 20, 21]. The Google Chrome Web Browser and Adobe have been successfully relying on isolation in the form of sandboxes ([4, 5]) to isolate non-trusted code remotely provided to users for execution on their computers. Isolation has proved to be a very effective mechanism and could play a vital role in protection of malicious activities in hardware. The goal of this tutorial is therefore to provide users a tool for seamless design and deployment of efficient hardware isolation techniques to address security concerns in FPGA-based systems.

The remainder of this section discusses current techniques that are employed to enforce hardware isolation on FPGAs. This list is by no means exhaustive and it is beyond the scope of this paper to discuss these techniques in greater details. The interested reader is encouraged to refer to cited works.

3.1 Reference Monitors

In this approach, a hardware module or a microcontroller based firmware-upgradable module is used to monitor and enforce authorized sharing of system resources among cores. Memory-access security policies are expressed in a specialized language, and a compiler translates these policies directly to a circuit (or a microcontroller) that enforces the policies. The circuit is then loaded onto the FPGA along with other components of the system.

There are many research efforts with some relation to this approach, but to the best of our current knowledge, works in [17] and [16] are the only work with similar goals that come close to ours here. In their work, they designed and implemented a reconfigurable reference monitor (RM) which implements an access control list (ACL). They then integrated the reference monitor into the on-chip peripheral bus (OPB) and used it to regulate access to the memory and peripherals. Memory and peripherals accesses go through a reconfigurable monitor's access control list [3]. The access control list associates every object (memory ranges for ex.) in the system with a list of principals (IP cores) with the rights of each principal to access the object [3]. In their implementation, each object access has to be computed by the reference monitor's ACL at runtime. The decision is either granted access or denied access.

Since this access model can create potential memory performance issues in large memory applications, they proposed a mechanism in which a buffer is used to hold the data until the ACL grants approval of the legality of the request [18]. Figure 3 illustrates the implementation. For example, in case of a write, the data to be written is stored in the buffer until the ACL grants the approval, at which time the write request is sent to the memory [18].

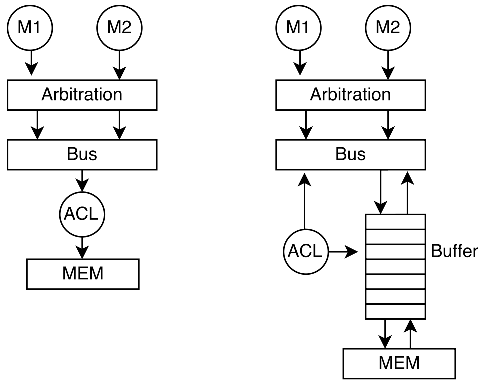

Figure 3: Access Model Implementation
On the left, there's no "caching" mechanism. On the right, a buffer is used to hold the data while access rights are being looked up.

Authors in [15] observed in a Huffmire et al. access model implementation, if you had consecutive and repeated memory access from the same "principal" to the same "object", each one of these requests would still have to go through the reference monitor's computation. This can potentially create performance issues in large designs. These issues can be avoided by adding the capability to remember access decisions. Authors in [15] built upon this observation and proposed an improved implementation of this architecture by adding capability to remember access decisions to allow them to be administered at run-time without re-computations. Figure 4 shows their proposed access model.

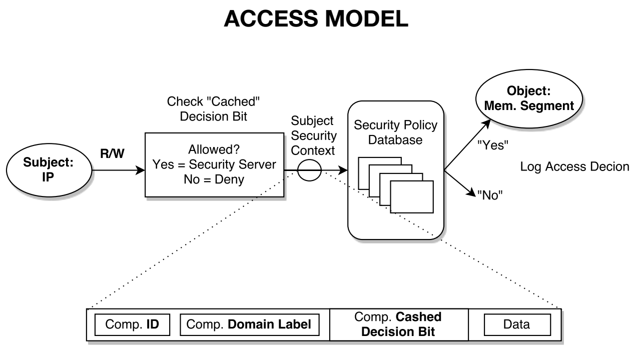

Figure 4: Improved Access Model Implementation

Both of these techniques follow a similar design flow. Systems components are defined and implemented using hardware synthesis tools such as Xilinx's Vivado or Xilinx's XPS. The hardware-enforced ACL core is generated by some policy compiler and it's then integrated with the rest of the system components similar to adding a standard custom IP to your design [18], [16]. In some instances, however, there can be compatibility issues. Sometimes standard integration of the ACL core can be complicated with the fact that some security-relevant attributes may not be directly available as parameters or easily derived from available parameters. For example, the current version of the AXI Interconnect IP core API available in Vivado 2016 does not present a component ID as a parameter. In situations like these, it's up to the application designer to build their own custom OPB (or AXI interconnect) IP which directly integrates this ACL security functionality.

Authors in [11], [24], [12] observed that the above described techniques assume a "trusted" reference monitor to enforce the security policies and with no mechanism through which the platform itself can authenticate the authority of the reference monitor before it administers an access policy decision. The security concern here is that an attacker could conduct a man-in-the-middle attack on the system by inserting a malicious circuit inside the only authority entrusted with administrating shared resource access decisions. To mitigate this, they proposed an improved implementation of the reference monitor which combines the monitoring approach with a "proof-carrying hardware" concept. Their approach consisted of using a consumer-producer approach, where a consumer specifies a desired functionality of the memory access monitor and sends this specification to the producer and the producer synthesizes this information into a bitstream [11]. The producer re-extracts the logic function from this bitstream and, together with the original specification, computes some miter function (which outputs an error flag if the specification and implementation differ for at least one input vector) [24]. This proof of reference monitor correctness is then generated along side with the bitstream and is sent to the consumer. The latter verifies the proof and, in case of success, partially reconfigures the monitor with security policies [24]. The consumer verifies the proof of correctness from the producer by extracting the monitor's logic function from the bitstream and forms a miter in conjunctive normal form in the same way as the producer, but with the original specification. This new miter is compared to the producer's miter and if they both match, the implementation is accepted and the monitor is accepted. The monitor is rejected if the the miters do not match [24], [12].

3.2 Secure Enclave: Zynq FPGA TrustZone

In this approach, systems developers rely on an operating system and some hardware-level enforcement mechanisms to provide system security services. Security features provided include, but are not limited to, confidentiality and integrity protection of the external memory containing the sensitive application code and data.

A recent example of this approach is the ARM TrustZone security architecture currently available to Zynq-7000 SoC. The ARM TrustZone architecture makes trusted computing within the embedded world possible by establishing a trusted platform, a hardware architecture that extends the security infrastructure throughout the system design. Instead of protecting all assets in a single dedicated hardware block, the TrustZone architecture runs specific subsections of the system either in a "normal world" or in a "secure world." [27].

In the Zynq-7000 AP SoC, a normal world is defined as a hardware subset consisting of memory regions, L2 cache regions, and specific AXI devices. The Zynq-7000 AP SoC supports ARM TrustZone technology in both the PS and PL domains of the device. The PS provides a set of configuration registers related to TrustZone support for custom IPs. These configuration registers are then dynamically programmed by the software during execution. All slave IP cores instantiated in the logic can be individually assigned a Secure or Non-Secure designation. For Xilinx slave IP cores, Secure/Non-Secure configuration can be statically designated at the AXI interconnect level during system creation process [26], [27].

Figure 5 shows a simplified example of how to secure your design sensitive components from illegal hardware access using Zynq-7000 AP SoC TrustZone technology.

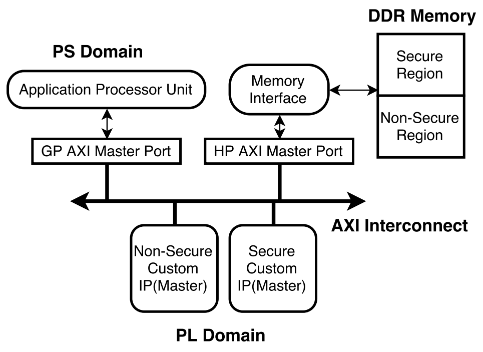

Figure 5: Improved Access Model Implementation

In this example, on the hardware level the PL domain uses two custom master IP cores:

- **Secure Custom IP:** This performs Secure read/write transactions to DDR using the HP slave port in the PS. It is also connected to the Application Processor Unit (APU) via a GP master port for register configuration.
- **Non-Secure Custom IP:** This is used to perform read-/write transactions from/to the PS DDR upon requests from the external world. It performs Non-Secure read transactions to DDR using an HP slave port. Configuration of this IP is done by the PS processor through a GP master port.

Upon power-on, the PS processor initializes both custom IPs, configures registers to establish Secure and Non-Secure regions in DDR memory, and transfers private data/instructions to the Secure DDR region. After initialization and upon receiving a data request from the external interface, the Non-Secure custom IP first copies the data from the external interface to the Non-Secure region of DDR memory. It then interrupts the PS processor, which commands the Secure custom IP to perform the appropriate data computations using its private data/instructions in conjunction with the data just copied to the Non-Secure region of DDR memory. After computation is completed, Secure custom IP puts the computed data into the Non-Secure DDR region and interrupts the PS processor to command the Non-Secure custom IP to start transferring data from the requested location [27].

On the software level, application processes (and their IPs) with non-secure status designation execute in memory space separated from processes with secure status designation. When a user process running in the Non-Secure world requires Secure execution, it makes a request to the Non-Secure kernel to enable the TrustZone Secure Monitor to transfer execution of the process to the Secure world. The Secure Monitor mode links the two zones and acts as a gatekeeper to manage program flow between them [26], [27]. To ensure integrity of the TrustZone software, Zynq-7000 provides a secure boot flow; where the on-chip BootROM code starts the whole security chain by ensuring that first-stage bootloader (FSBL) is signed and verified [27].

3.3 Hardware Sandboxing

The approach of utilizing hardware sandboxes for isolating and identifying potential malicious IP-internal circuits, as presented by the authors of [23], provides a flexible design process for systems designers and integrators. Similar to discussed approaches, a sandbox isolates IP by partitioning the hardware design to a trusted secure region and a non-trusted environment contained within the sandbox. It supports the following security policies: Isolation policy that limits the IP to only allowable interactions, Containment policy that minimizes the damage caused by a malicious component, and Integrity policy that enforces authorized data accessibility and modifications. These policies are implemented through two distinct but complementary mechanisms: *Complete Mediation, and Resources Virtualization.*

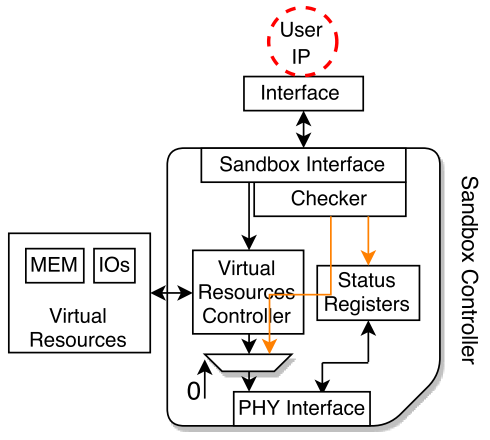

Figure 6: Hardware Sandbox Structure

Hardware sandboxing allows the placement of interface-level behavioral monitors, collected in a *Checker* component that implements the *Complete Mediation* mechanism by checking every interaction made by the user IP to the system and only authorizing allowable interactions as specified by the IP interface specification.

The *Sandbox controller* implements the *Resources Virtualization* mechanism by providing both virtual memory buffers (local to the user IP) and virtual IO access for critical resource protection. It also generates a *Virtual Resources Controller* in case there are more virtual resources competing for a single physical resource. *Status registers* are used to monitor and log the user IP security status.

This is useful in that a subverted IP can't launch a denial-of-service attack by repeatedly making illegal requests. Also, this can be used to log the user IP if it triggers a Trojan at run-time for example. This information can then be used to blacklist some vendors from the integrator's list of contractors. This structure as presented by the work of [23], is shown in Figure 6.

4 SECURITY CRITICAL EXAMPLE APPLICATION

In this section, we introduce an example application to demonstrate the various hardware isolation methods discussed above.

4.1 Overview

We intend to demonstrate the steps required to ensure the protection of a general security-critical system design with an SoC system providing reconfigurable fabric along with a co-processor. The discussed isolation methods enforce security at the OS/PS level down to the hardware IP interface level. Thus, our demonstration application includes process level applications running on the processing system while hardware is utilized for acceleration cores. The system requires a security critical application such an application requiring sensitive keys to be handled at the software level while relying on hardware for accelerating cryptographic functions.

4.1.1 Secure Echo Server Process: We utilize an echo server that encrypts client communications with a session key established through a simplified TLS/SSL handshake. This system setup assumes the server to be run on the SoC device targeted for protection while the client is run on a remote machine. The SoC hosting the echo server is designed to run on a Zynq SoC, with the zynq processor hosting a Linux OS and utilizing FPGA area for cryptographic functions. The echo server process and complementing client process are implemented as Python programs. We used Xilinx's Petalinux tool for generating the design images required to boot a Linux OS and mount the filesystem. We implemented our system on a ZC706 board (XC7Z045 FFG900-2 AP SoC).

4.1.2 Hardware Accelerated Encryption Engine: The handshake requires the use of both symmetric-key and public-key algorithms, as a symmetric-key (used as session key) is generated from utilizing a asymmetric-key to encrypt and publically transmit the session key. AES and RSA are commonly used to meet these requirements. As these can be computationally expensive with larger key sizes, one standard application of hardware acceleration is for performance boosts with cryptographic functions. To provide our demo system with a hardware component, an AXI-bus enabled IP core is used to accelerate the AES and RSA algorithms used by the echo server process.

Upon requiring the AES or RSA cores during execution, data is first written to the appropriate input data registers for each IP. After a successful write of input data, an enable signal is written to a control register to signal the IP core to perform the encryption/decryption. A set of output data registers are assigned to each core along with a status register to inform the software process of a complete computation.

The hardware cores were implemented as Vivado IP cores and exposed by supplying an AXI-bus interface for reading inputs and writing outputs to DDR memory. This is accessible by default from the OS as Petalinux images include kernel drivers for exposing DDR memory under the OS's '/dev/mem' filepath. The memory blocks for the cryptographic core are accessible via their device-tree specified address.

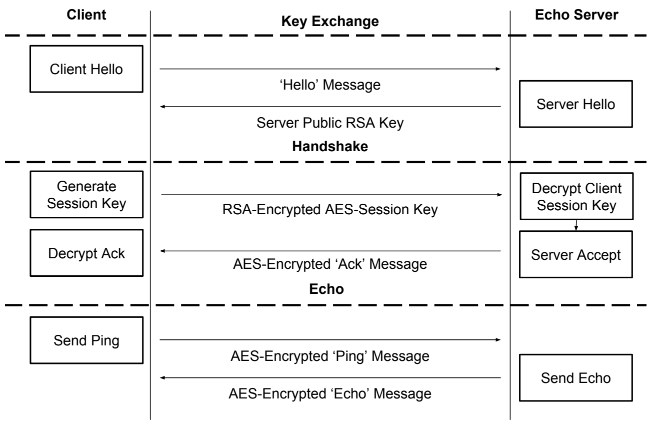

Figure 7: Baseline Behavior for Client and Echo Server
In its initial state, the server process listens on a specified network IP address and port awaiting a client connection. Upon receiving a connection, a handshake is initiated as the server immediately shares an RSA public key. A session key computed by the client is received by the server encrypted with its public RSA credentials. Deciphering this session key allows subsequent communications to be secured, namely the server's acknowledgement of a successful handshake, the client's message, and finally the server's echoed message.

4.2 Security Vulnerabilities

Our application was designed to include vulnerabilities that could be found in designs that include software processes and hardware accelerators handling sensitive information without security measures. An operating system lacking the ability to restrict executions on sensitive data to isolated processor environments and verify process permissions for secured memory space creates potential points of attack for malicious processes. With the hardware design community's growing adoption rate of third-party IP cores, there are a multitude of infiltration points within the development and manufacturing process with which malicious parties have been able to exploit.

To exploit the vulnerabilities mentioned, we have included a non-secure process and hardware trojans for the purpose of demonstration and providing a metric of system security strength. It should be recognized that the server/client handshake is missing vital components of a true TLS/SSL scheme such as certificate based verifications performed by both client and server. However, the scope of this work omits the undertaking of securing the network layer.

4.2.1 Software Threat Insertion: For the purpose of this tutorial, we wrote a (non-secure) process thread as part of our application software to randomly read memory regions associated with cryptographic key management code variables and data structures. The non-secure process conducts unauthorized code injection on these privileged locations and tries to gain privileged secure execution status (privilege escalation attack).

4.2.2 Hardware Threat Insertion: Our encryption engine is a prime candidate for inserting a malicious circuit as it will have access to sensitive encryption key data. As our application performs a simple handshake and utilizes both AES and RSA algorithms, we can leverage the Trust-Hub.org repository of hardware trojans for AES128 and BasicRSA. Both AES128 and BasicRSA trojan classes contain variations of activation methods, from always-on trojans to internal condition triggers. We sample the set of available AES128 and BasicRSA trojans and select the variations shown in in Table 1. Selecting any AES and RSA pair from this subset provides our demo system with a range of malicious behaviors originating from hardware circuits that are typically difficult to detect without the application of some security methods.

Table 1: Attributes of Trust-Hub Trojan Subset

Design	Trojan Class	Activated?	Trigger	Effect
AES-T400	Info Leak	Yes	Input Value	Key Leak via AM Transmission
AES-T500	DoS	Yes	Input Sequence	Power Draw Increase
AES-T600	Info Leak	Yes	Input Value	Key Leak via Side Channels
AES-T700	Info Leak	Yes	Input Value	Key Leak via Side Channels
AES-T800	Info Leak	Yes	Input Sequence	Key Leak via Side Channels
AES-T900	Info Leak	Yes	Internal Counter	Key Leak via Side Channels
AES-T1000	Info Leak	Yes	Input Value	Key Leak via Side Channels
BasicRSA-T100	Info Leak	Yes	Input Value	Key Leak via Protocol
BasicRSA-T200	Info Leak	Yes	Input Value	Data Leak via Protocol
BasicRSA-T300	Info Leak	Yes	Internal Counter	Key Leak via Protocol
BasicRSA-T400	DoS	Yes	Internal Counter	Data Leak via Protocol

4.3 Addressing Vulnerabilities with Isolation

4.3.1 Reference Monitor and TrustZone: We assume no a priori knowledge of specific vulnerabilities within the application code running on the FPGA SoC. All we know is that we are tasked to provide a secure implementation of the application. TrustZone provides us with the ability to create memory regions that are isolated from all other code in the system (including privileged code such as OS kernel). Depending on the size of the code that's going to execute in this secure regions, we may have to segment these memory regions into smaller segments whose utilization and access permissions can be tracked through a hardware-implemented reference monitor. Segmenting these isolated secure regions is important since it allows us to create a tiered trust model in our system. This way we can guarantee confidentiality of our application software most sensitive data (cryptographic keys) by isolating it from all privileged code.

4.3.2 Hardware Sandboxing: The protections put in place with reference monitors and Trustzone are limited as they do not consider malicious hardware components interacting with secure processes. Though the process execution environment may be secured with relevant memory blocks protected, an assumed-secure IP core could house a malicious hardware trojan. To complement the methods discussed for secure process execution, we propose the use of sandboxing to extend policy-driven and isolation-based security to the programmable logic of an SoC. Thus far, we have addressed vulnerabilities in all components of our demo systems with the exception of our cryptographic hardware cores. With the security-critical server/client processes utilizing hardware cores, it is essential that the IP are behaving as expected. Hardware sandboxing provides a way to safely integrate nontrusted IP into a secure system as we are able to monitor all nontrusted IP interface interactions and isolate unexpected interactions with secured resources.

5 IMPLEMENTING ISOLATION

In this section, we discuss implementation details regarding the steps of applying the isolation methods to our security critical application. Each of the methods has unique strengths and a more comprehensive security profile can be realized upon implementing each in conjunction.

To discuss the implementation in a meaningful way, we discuss each of the isolation methods in the order they would appear in a bottom-up system development process. We begin with isolating the cryptographic accelerators in hardware and proceed to configuring a secure execution environment. With our secure process utilizing hardware-accelerated encryption, CAPSL is utilized to secure the encryption engine and enforce the integrity of its behavior. This system suits the implementation of the discussed isolation methods with the inclusion of points of attack in both software and hardware.

We caution the reader that implementation details discussed in this section are only applicable in the context of tools listed. Finer details regarding some aspects of our implementation may change as you employ a different set of tools.

5.1 Hardware Level

5.1.1 CAPSL. To enforce a deeper security profile in our system design, we utilize hardware sandboxing to isolate the untrusted encryption cores. We leverage the secure development tool presented in [23], the Component Authentication Process for Sandboxed Layouts (CAPSL), as it supplies automation for generating hardware sandboxes, allowing for a streamlined integration of questionable components into systems.

5.1.2 Design Flow. A specification defining the interface and its behavior is required for each IP in order to formally model the IP and generate automata objects of each policy. The set of automata produced from all policies is used as a behavioral monitor within the sandbox. The specification enables the flexibility to define behavior at varying levels of abstraction and varying degrees of completeness. To elaborate, specification might contain partial interface definitions and/or behaviors abstracted through additional computation logic. To capture security properties, CAPSL adopts the Interface Automata (IA) formalism of De Alfaro and Hetzinger [10] along with a subset of the Properties Specification Language (PSL), Sequential Extended Regular Expression (SERE) [14]. As a baseline specification, only interface layouts are required (IP interface inputs, outputs). Beyond this, any combination of IA-based descriptions of allowed behavior, explicitly denied interactions defined as SERE statements, and computational logic is allowed.

```
// AES128.config - CAPSL Configuration File
//   IP Targets: Trust-Hub AES Trojans T400, T500, T600, T700, T800, T900, T1000
AES.def{ // AES 128 Interface Layout and Expected Interactions
  reset       : input,
  key         : input-vector(AESKeySize),
  state       : input-vector(AESKeySize),
  out         : output-vector(AESKeySize),
  transitions{} // No transitions are required
}
logic.def{ // Additional Computational Logic and SERE Restrictions
  AESKeySize = 128, // AES KeySize
  logic0 : state == x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF", // T400, T600 trigger input
  logic1 : state == x"00112233445566778899AABBCCDDEEFF", // T700, T1000 trigger case

  logic2 : xmit_dataH == x"3243F6A8885A308D313198A2E0370734", // T500, T800 trigger sequence
  logic3 : xmit_dataH == x"00112233445566778899AABBCCDDEEFF",
  logic4 : xmit_dataH == x"00000000000000000000000000000000",
  logic5 : xmit_dataH == x"00000000000000000000000000000001",

  counter1 : counter { // T900 Trigger counter
    on: event(out),
    start: x"00000000000000000000000000000000",
    end: x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
  }

  prohibited{ // Illegal conditions (SERE)
    {logic0 | logic1 | counter1}, // Restrict T400, T600, T700, T900, T1000 trigger cases
    {logic2 ; logic3 ; logic4 ; logic5} // Restrict T500, T800 trigger sequence
  }
}
```

Figure 8: AES128.config - CAPSL Specification

Implementing a hardware sandbox for our demo system requires the specification of our untrusted hardware components. The specification files for AES and RSA cores are detailed in Figure 8 and Figure 10, respectively. Along with the required interface layout definition, the specifications for both AES and RSA utilize SERE statements in conjunction with addition computational logic to provide explicitly denied interactions. More specifically, the specifications are utilizing additional logic for abstracting the SERE specifications for detecting known trojan triggers. The RSA specification also utilizes the IA-based description of allowed behavior to define how the IP shares its current state. With our specification files providing policies to isolate undesired behavior, we can simply execute CAPSL to obtain a sandbox IP containing our hardware cores. Assuming the specifications are located within a single directory, CAPSL is run with the following:

```
./CAPSL <Config Directory Path>
```

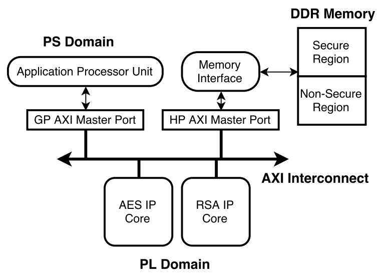

Figure 9: Demo System Block Design

Taking a brief look at CAPSL's internal generation process, the IP models resulting from specification are first subjected to an optimization phase. The sandbox design leverages the IA composition operation to provide compatible and secure interfaces between the IP and the rest of the system through the sandbox. Applying the operation requires the environment (the IP in our case) not to perform illegal actions, i.e. actions that the sandbox and therefore the rest of the system doesn't expect. This means that

```
// BasicRSA.config - CAPSL Configuration File
//   IP Targets: Trust-Hub BasicRSA Trojans T100, T200, T300, T400
BasicRSA.def{ // Basic RSA Interface Layout and Expected Interactions
  ds          : input,
  ready       : input,
  reset       : input,
  inExp       : input-vector(RSAKeySize),
  inMod       : input-vector(RSAKeySize),
  inData      : input-vector(RSAKeySize),
  cypher      : output-vector(RSAKeySize),
  transitions{
    s0:ds>s1, // ds asserted to signal inputs ready
    s1:!ready>s2, // ready goes low
    s2:!ds>s3, // ds goes low
    s3:ready>s0 // ready asserted when finished and prepared for next inputs
  }
}
logic.def{
  RSAKeySize = 32, // RSA KeySize
  logic0 : cypher == inExp, // Checking for plaintext key leaks (T100, T200, T300)
  logic1 : inData == x"44444444", // Checking for T100 trigger case
  logic2 : inData == x"01FA0301", // Checking for T200 trigger case

  counter1 : counter { // T300, T400 Trigger counter
    on: { logic3 : ds == '1' },
    start: x"00000000",
    end: x"FFFFFFFF"
  }

  prohibited{ // Illegal conditions (SERE)
    {logic0}, // Restrict key leaks of T100, T200, T300
    {logic1 | logic2 | counter1}, // Restrict T100, T200, T300, T400 trigger cases
  }
}
```

Figure 10: BasicRSA.config - CAPSL Specification

interfaces are assembled only if they are compatible, resulting in a new composed interface. With all policies represented as Interface Automata with CAPSL, the composition operation can supply an avenue for reducing the policy automata set size through merging automata with consideration for the environmental assumptions of it.

The resulting optimized set of policies is to be translated to a reference monitor. Since CAPSL translates from a formal model, it is possible to insert a variety of target IP implementations for generating the sandbox as an IP. Using a VHDL flow as a result of leveraging Trust-Hub's RTL level trojan implementations, the "Checker" is generated by one-hot encoding of the monitoring automatons as VHDL statements which capture the expected behavior of the IP. The checker module is combined together with virtual resources (BRAMs, SLRs, etc.) to generate the core logic which governs the sandbox. The generated core logic is then combined with a sandbox controller (acting as a router for all signals between interfaces and checker) to conclude the sandbox generation. The controller consists of the sandbox interface (composed with the non-trusted IP interface), physical interface, some status registers, and a multiplexer which acts as a switch to either allow IP interactions to continue or to invalidate them.

Upon finishing, the tool outputs the components of the sandbox as VHDL source files, centralized within the default *Outputs* directory local to the execution path. which can be immediately imported to design and synthesis tools for integration into the target system. For our demonstration system, we utilize Xilinx Vivado to design and synthesize the block design as seen in Figure 9. In preparation for TrustZone, this diagram updates the improved access model shown in Figure 5. Using a Xilinx ZC706 SoC, with dual Zynq ARM Cortex A9 processors, we allot one processor for hosting an OS, configure the ethernet physical interface, and tie the hardware accelerator cores to the PS via an AXI Interconnect.

Since TrustZone is either implemented or not (it's not "enabled"), when generating the hardware description files we are required to specify the security status of all AXI components. Xilinx Vivado tool provides the ability to set the NS (bit specifies if whether the bus access is from a secure component or a "normal" component) when constructing the hardware system.

By default, TrustZone enabled CPUs boot in the "secure" world. So it's important we specify and mark non-secure regions in the memory *before* we generate the bootloader. This step is also done when generating the hardware system. Specifically, this is done by writing into `TZ_DDR_RAM` register. Each bit in `TZ_DDR_RAM` represents the TrustZone status for a 64 MB section n at nMB, where 0 corresponds to Secure, reset value and 1 denotes Non-secure.

5.2 Software Level

To bring up the software (bootloader, embedded Linux kernel image, and file system), we employ the `PetaLinux tools` to generate the `boot.bin` file and the `image.ub` file (interested readers can refer to [25] for all PetaLinux commands needed for these steps.). Involved steps are the following:

(1) Export the hardware created in section 5.1 (the `bit` file) to Xilinx Vivado SDK tool to generate the hardware description file (`HDF` file). The latter contains information regarding the target board and the hardware implementation.
(2) Create a Zynq PetaLinux project that targets Zynq processor. In this step, you provide the `HDL` file (generated in the previous step).
(3) Build the PetaLinux project. This steps generates the device tree, the Linux kernel image, the first stage bootloader (`FSBL`), and the file system (this is a volatile file system so you may want to provide your own file system).
(4) Use Petalinux to create the `boot.bin` file by issuing the command `petalinux-package --boot --fpga <FPGA bitstream> --u-boot`. The tool will generally use the FSBL generated in the previous step automatically. But you are still expected to provide the path to the `.BIT` file.
(5) Load the `boot.bin`, `image.ub` (also generated in (3)), the file system, and the drivers files to an SD card and boot the system.

Layered security (i.e. combining multiple security mechanisms to protect user's data) is generally recommended to further strengthen the system security envelop [3], [22]. So, securing our device boot process would normally be additional point of focus for overall device system security. But, for the scope of this tutorial, we assume the system boots in the trusted state and the system software is ready to start the example application.

As previously described, the example application software implements a simple client and echo server. All secure computations are handled within the "secure world" context while other system tasks (application logs,non-critical asserts, our malicious thread, etc.) runs in the "normal" world. Context switch between "secure" and "normal" world is handled by the CPU's Monitor mode through `Secure Monitor Call (SMC)` exceptions. All unauthorized attempts by our malicious process threads to access sensitive data was successfully captured through the error handler. The latter is an interrupt service routine (`ISR`) which is called when an AXI bus error response is generated.

6 CONCLUSION

This tutorial explored the use of isolation methods to implement security in a general SoC architecture. We presented a number of techniques and discussed the strengthening of system security implementations with a hybrid approach. As the presented isolation techniques each individually address security in somewhat disjoint domains, we demonstrate a more effective secure architecture utilizing isolation schemes at both the hardware and software levels. We develop a general SoC design executing on sensitive data that also contains a number of exploits targeting common vulnerabilities. We presented the design steps necessary to implement our security critical application with secure execution environments (TrustZone) and hardware sandboxing (CAPSL). Upon implementation of the secure design, the security policies put in place were verified against the malicious process and trojans injected to our design. We present this demonstration as a means to promote the exploration and use of comprehensive isolation methods with coverage of all SoC levels.

REFERENCES

[1] Amazon. 2017. Amazon EC2 F1 Instances. (2017).

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 689–703.

[3] Abhishek Basak, Swarup Bhunia, and Sandip Ray. 2015. A Flexible Architecture for Systematic Implementation of SoC Security Policies. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '15)*. IEEE Press, Piscataway, NJ, USA, 536–543. http://dl.acm.org/citation.cfm?id=2840819.2840894

[4] Adobe Blog. Security@Adobe. http://blogs.adobe.com/security/. (????).

[5] Chromium Blog. 2008. A new approach to browser security: the Google Chrome Sandbox. https://blog.chromium.org/2008/10/new-approach-to-browser-security-google.html. (Oct 2008).

[6] C. Bobda, T. J. L. Whitaker, C. Kamhoua, K. Kwiat, and L. Njilla. 2017. Synthesis of hardware sandboxes for Trojan mitigation in systems on chip. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 172–172. https://doi.org/10.1109/HST.2017.7951836

[7] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2014. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines* (2014), 109–116.

[8] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the Cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14)*. ACM, New York, NY, USA, Article 3, 10 pages.

[9] Lance Cottrell. 2017. Isolation Technology Protects Against Risky Cyberenvironments. https://ntrepidcorp.com/cyber-security/isolation-technology-protects-against-risky-cyberenvironments/. (Feb 2017).

[10] Luca de Alfaro and Thomas A. Henzinger. 2001. Interface Automata. *SIGSOFT Softw. Eng. Notes* 26, 5 (Sept. 2001), 109–120. https://doi.org/10.1145/503271.503226

[11] S. Drzevitzky. 2010. Proof-Carrying Hardware: Runtime Formal Verification for Secure Dynamic Reconfiguration. In *2010 International Conference on Field Programmable Logic and Applications*. 255–258. https://doi.org/10.1109/FPL.2010.59

[12] S. Drzevitzky, U. Kastens, and M. Platzner. 2009. Proof-Carrying Hardware: Towards Runtime Verification of Reconfigurable Modules. In *2009 International Conference on Reconfigurable Computing and FPGAs*. 189–194. https://doi.org/10.1109/ReConFig.2009.31

[13] Fireglass. 2016. Fireglass Emerges From Stealth with a Military-Grade, Enterprise-Ready Threat Isolation Platform to Put an End to Security Challenges. (Feb 2016).

[14] Ziv Glazberg, Mark Moulin, Avigail Orni, Sitvanit Ruah, and Emmanuel Zarpas. 2007. *PSL: Beyond Hardware Verification.* Springer Netherlands.

[15] F. Hategekimana and C. Bobda. 2017. Applying the Flask Security Architecture to Secure SoC Design. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 198–198. https://doi.org/10.1109/FCCM.2017.28

[16] Ted Huffmire, Brett Brotherton, Nick Callegari, Jonathan Valamehr, Jeff White, Ryan Kastner, and Tim Sherwood. 2008. Designing Secure Systems on Reconfigurable Hardware. *ACM Trans. Des. Autom. Electron. Syst.* 13, 3, Article 44 (July 2008), 24 pages. https://doi.org/10.1145/1367045.1367053

[17] T. Huffmire, B. Brotherton, T. Sherwood, R. Kastner, T. Levin, T. D. Nguyen, and C. Irvine. 2008. Managing Security in FPGA-Based Embedded Systems. *IEEE Design Test of Computers* 25, 6 (Nov 2008), 590–598. https://doi.org/10.1109/MDT.2008.166

[18] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine. 2007. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems. In *2007 IEEE Symposium on Security and Privacy (SP '07).* 281–295. https://doi.org/10.1109/SP.2007.28

[19] Intel. 2017. Report: Intel Inside New Audi Autonomous Car System. (2017).

[20] Franklyn Jones. Cyber Security for Credit Unions: Managing the Unique Challenges. https://www.bankinfosecurity.com/webinars/how-isolation-technology-protects-cus-from-cyber-attacks-w-575. (????).

[21] John P. Mello Jr. 2015. Is Isolating the Internet Key to Bulletproof Security? http://www.technewsworld.com/story/82245.html. (July 2015).

[22] S. Ray and Y. Jin. 2015. Security policy enforcement in modern SoC designs. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* 345–350. https://doi.org/10.1109/ICCAD.2015.7372590

[23] T. J. L. Whitaker and C. Bobda. 2017. CAPSL: The Component Authentication Process for Sandboxed Layouts. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI).* 409–414. https://doi.org/10.1109/ISVLSI.2017.78

[24] T. Wiersema, S. Drzevitzky, and M. Platzner. 2014. Memory security in reconfigurable computers: Combining formal verification with monitoring. In *2014 International Conference on Field-Programmable Technology (FPT).* 167–174. https://doi.org/10.1109/FPT.2014.7082771

[25] Xilinx. 2014. PetaLinux Getting Started. (2014).

[26] Xilinx. 2014. Programming ARM TrustZone Architecture on the Xilinx Zynq-7000 All Programmable SoCs. (2014).

[27] Xilinx. 2014. TrustZone Technology Support in Zynq-7000 All Programmable SoCs. (2014).