Hardware Isolation Mechanisms for Security Improvement in FPGAs

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 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 paper, we present a tutorial on 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.

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 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. 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 recent announcement of providing cloud compute instances with FPGAs (EC2 F1).

FPGA security continues to be a strong area of concern. Specifically, current FPGA platforms do not possess any hardware primitives that would allow support for multi-tenancy while enforcing proper domain isolation[STILL WORKING ON INTRODUCTION].

2 HARDWARE ISOLATION MECHANISMS

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.

2.1 Hardware-enforced Access Control Lists

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 [CITATION] and in [CITATION] 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 [3]. Figure 1 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 [3].

Authors in [CITATION] observed that in 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 [CITATION] 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 2 shows their proposed access model.

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 [CITAT],[CITAT]. In

ICDSC'11, Stanford University, CA, USA

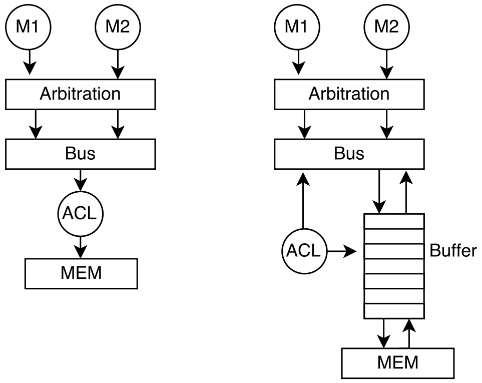

Figure 1: Access Model Implementation. On the left, there's no "caching" mechanism. On the right, a buffer is used to hold the data while access rights are being looked up.

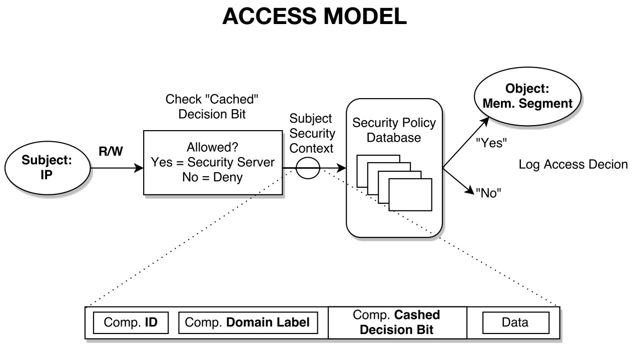

Figure 2: Improved Access Model Implementation.

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 [CITATION], [CITATION] 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. 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). 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. 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.

2.2 OS-Enforced Security: Zynq FPGA TrustZone

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

A recent example of this approach is the ARM TrustZone security architecture currently available to Zynq-7000 SoCs. In this example, the Xilinx Zynq-7000 processor system supports ARM TrustZone technology in both the PS and PL domain. 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 a "secure world."

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.

Figure 1 shows a simplified example of how to secure your design sensitive components from illegal hardware access using Zynq-7000 AP SoC TrustZone technology

In this example, also available in [], 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

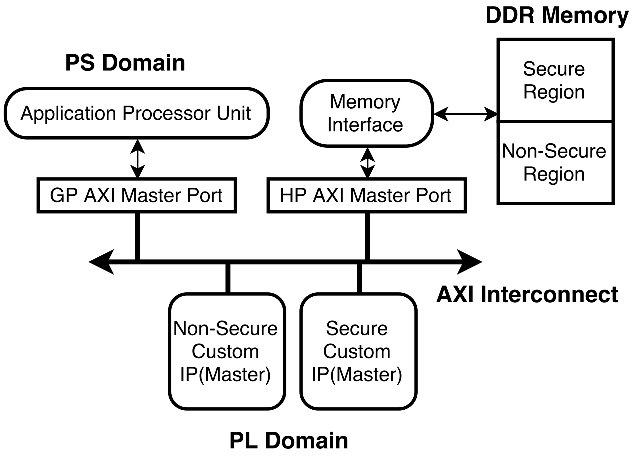

Figure 3: Improved Access Model Implementation.

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.

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.

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.

2.3 CAPSL: Automatic Generation of Hardware Sandboxes

The approach of utilizing hardware sandboxes for isolating and identifying potential malicious IP-internal circuits provides a flexible design process for systems designers and integrators. Similar to discussed approaches, a sandbox isolates IP by partioning the hardware design to a trusted secure region and a non-trusted environment contained within the sandbox. Hardware sandboxing allows the placement of an interface-level reference monitor to ensure the integrity of non-trusted IP interactions. In addition, virtual resources can be provided to protect critical system resources. The sandbox approach allows unrestricted access to resources used by the trusted IP, while containing potential negative effects of untrusted components behind a behavioral monitor and resource virtualization.

CAPSL, the Component Authentication Process for Sandboxed Layouts supplies automation for generating hardware sandboxes, allowing for a streamlined integration of questionable components into secure systems. The design flow is summarized in the following tasks:

2.3.1 Construct Internal IP Interface Models: 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 defintions and/or behaviors abstracted through additonal computation logic. To capture security properties, CAPSL adopts the Interface Autoamta (IA) formalism of De Alfaro and Hetzinger [CITATION NEEDED] along with a subset of the Properties Specification Language (PSL), Sequential Extended Regular Expression (SERE) [CITATION NEEDED]. As a baseline requirement, 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.

2.3.2 Optimize Models: We chose IA as our internal model due to its capability to handle *compatibility* and *refinement*. While compatibility insures that two communicating components do not perform illegal action, i.e one automaton produces an input that the other cannot consume, refinement allows breakdown within component boundary while still maintaining the compatibility. The sandbox design leverages these two properties to provide compatible and secure interfaces between the IP and the rest of the system through the sandbox. Applying the compatibility concept of IA 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 interfaces are assembled only if they are compatible, resulting in a new composed interface. With all policies respresented as Interface Automata, 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.

2.3.3 Generate/Package Sandbox IP:. The resulting optimized set of policies can now be translated to a reference monitor. When translating from a formal model, it is possible to insert a variety of target IP implemetnations for generating the sandbox as an IP. Initial development of CAPSL produced the sandbox as Vivado IP implemented as VHDL. CAPSL, using a VHDL flow, generates a "checker" module. The module is generated by one-hot encoding our 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 (routing 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.

3 SECURITY CRITICAL APPLICATION EXAMPLE

In this section, we introduce an example application to demonstrate the various hardware isolation methods discussed above. Each of the methods has unique strengths and a more comprehensive security profile can be realized upon implementing each in conjunction. We propose a system that performs secure encryption processes (via Hardware-enfored Access Lists and TrustZone) alongside potentially malicious non-secured processes. 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 hardware isolation methods with the inclusion of points of attack in both software and hardware. Below, we detail the example application and address the security vulnerabilites with regard to the introduced isolation methods.

3.1 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.

3.1.1 *Secure Echo Server Process:*

We utilize an echo server that encrypts client communcations 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 Digiglent Zybo [IT MIGHT NOT BE BIG ENOUGH for AES + RSA]...

To further elaborate on our application, we begin with the echo server process. The 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. This system behavior as seen at the process level can be seen in [NEED DIAGRAM].

3.1.2 *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. The PS and PL interaction is detailed in [NEED DIAGRAM].

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. This will be revisited later in [POSSIBLY ANOTHER SECTION TO GO DEEP INTO DETAIL].

3.2 Security Vulnerabilities

Our application was designed to include vulnerabilities that could be found in designs that include software processes and hardware accelerators handling sensitve 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. It should be noted that it is recognized 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.

To exploit the vulnerabilities mentioned, we have included a non-secure process and hardware trojan for the purpose of demonstration and providing a metric of system security strength.

3.2.1 *Software Threat Insertion:*

[SECTION ABOUT A NONSECURE PROCESS INTENDED TO ACCESS SENSITVE DATA]

3.2.2 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.

3.3 Adressing Vulnerabilities with Isolation

An accepted paradigm of security for our proposed system would require secure execution for sensitive computations with dedicated hardware resources that are deemed secure. We address the system vulnerabilities as follows:

3.3.1 HACLs and TrustZone: [FESTUS]

3.3.2 Hardware Sandboxing: The protections put in place with HACLs 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 CAPSL to extend policy-driven and isolation-based security to the programmable logic of an SoC. We have addressed 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. By securing the cores within a sandbox, we are able to monitor all nontrusted IP interface interactions and isolate unexpected interctions with secured resources.

4 IMPLEMENTING ISOLATION

In this section, we discuss implementation details reagrding the steps of applying the isolation methods.

4.1 HACLs

4.2 TrustZone

4.3 CAPSL

As discussed in Section 2.3, the CAPSL design flow requires specifications for nontrusted IP (Cryptographic cores) purposed for defining interface connections and behavior. The IP models resulting from specification are then subjected to an optimization phase. The collection of models are translated to a reference monitor implementation following optimization. The final output is an implementation of the hardware sandbox packaged as an IP.

4.3.1 IP Specification: To utilize CAPSL's sandbox generation process, we must define the IP we wish to include. Due to the underlying modelling of the untrusted IP, specification at the interface level, or any abstraction of this, is required. That is, specification might contain partial interface defintions and/or behaviors abstracted through additonal computation logic. The specification files for AES and RSA cores are explained in Figure [RSA AND AES SPEC FILES EXPLANATION DIAGRAM].

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 additonal 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.

4.3.2 Model Optimization:

4.3.3 Sandbox Generation:

5 CONCLUSION

REFERENCES

[1] Nicola Dell, Trevor Perrier, Neha Kumar, Mitchell Lee, Rachel Powers, and Gaetano Borriello. 2015. Paper-Digital Workflows in Global Development Organizations. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work &#38; Social Computing (CSCW '15)*. ACM, New York, NY, USA, 1659–1669. https://doi.org/10.1145/2675133.2675145

[2] Tom Dietterich. 1995. Overfitting and Undercomputing in Machine Learning. *ACM Comput. Surv.* 27, 3 (Sept. 1995), 326–327. https://doi.org/10.1145/212094.212114

[3] F. Girosi, M. Jones, and T. Poggio. 1995. Regularization Theory and Neural Networks Architectures. *Neural Computation* 7, 2 (March 1995), 219–269. https://doi.org/10.1162/neco.1995.7.2.219

[4] Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay D. Shet. 2013. Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. *CoRR* abs/1312.6082 (2013).

[5] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov 1998), 2278–2324. https://doi.org/10.1109/5.726791

[6] Salim Ouchtati, Mouldi Bedda, and Abderrazak Lachouri. 2007. Segmentation and Recognition of Handwritten Numeric Chains 1. (2007).

[7] L. B. Saldanha and C. Bobda. 2015. A System on Reconfigurable Chip for Handwritten Digit Recognition. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 166–166. https://doi.org/10.1109/FCCM.2015.44

[8] L. B. Saldanha and C. Bobda. 2016. Sparsely connected neural networks in FPGA for handwritten digit recognition. In *2016 17th International Symposium on Quality Electronic Design (ISQED)*. 113–117. https://doi.org/10.1109/ISQED.2016.7479185

[9] Tom Schaul, Sixin Zhang, and Yann LeCun. 2013. No more Pesky Learning Rates. In *Proc. International Conference on Machine learning (ICML'13)*.

[10] J. Tang, C. Deng, and G. B. Huang. 2016. Extreme Learning Machine for Multilayer Perceptron. *IEEE Transactions on Neural Networks and Learning Systems* 27, 4 (April 2016), 809–821.

[11] Xilinx 2014. *ZYBO Reference Manual*. Xilinx. v6.

[12] Xuan Yang and Jing Pu. 2015. Multi-digit Recognition using Convolution Neural Network on Mobile. (2015). https://web.stanford.edu/class/cs231m/projects/final-report-yang-pu.pdf