ENABLING USER SPACE SECURE HARDWARE

By

AIMEE COUGHLIN

B.S. Computer Science, Colorado School of Mines, 2013

B.S. Electrical Engineering, Colorado School of Mines, 2013

M.S. Electrical Engineering, University of Colorado Boulder, 2015

A dissertation submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computer and Energy Engineering

2018

This dissertation entitled:
Enabling User Space Secure Hardware
written by Aimee Coughlin
has been approved for the
Department of Electrical, Computer and Energy Engineering

_____
Professor Eric Keller

_____
Professor Eric Wustrow

_____
Professor Sangtae Ha

_____
Professor Fabio Somenzi

_____
Professor Dirk Grunwald

Date_____

The final copy of this dissertation has been examined by the signatories, and we find that the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Coughlin, Aimee (Ph.D., Electrical Engineering, Department of Electrical, Computer and Energy Engineering)

Enabling User Space Secure Hardware

Dissertation directed by Assistant Professor Eric Keller

User space software allows developers to customize applications beyond the limits of the privileged operating system. In this dissertation, we extend this concept to the hardware in the system, providing applications with the ability to define secure hardware; effectively enabling hardware to be treated as a user space resource. This addresses a significant challenge facing industry today, which has an increasing need for secure hardware. With the ever increasing leaks of private data, increasing use of a variety of computing platforms controlled by third parties, and increasing sophistication of attacks, secure hardware, now more than ever, is needed to provide protections we need. However, the current ecosystem of secure hardware is fractured and limited. Developers are left with few choices of platforms to implement their applications and oftentimes the choices dont fully meet their needs. Instead of relying on manufacturers to make the correct design decisions and ensuring that these platforms are implemented correctly, we enable applications to define the exact secure hardware that it needs to protect itself and its data.

This vision leverages the emergence of programmable hardware, specifically FPGAs, to serve as the basis of user space secure hardware. The challenges of this, however, are that (i) sharing of FPGA resources among multiple applications is not currently practical, and (ii) the reprogrammability of FPGAs compromises the security properties of secure hardware. We address these challenges by introducing two systems, Cloud RTR and Software Defined Secure Hardware, which individually solve each challenge, and then combine these solutions together to realize the complete vision. Cloud RTR solves the first challenge by leveraging cloud compilation to allow for an

FPGA to be shared between applications, making hardware into a user space resource. SDSHW solves the second challenge by introducing a self-provisioning system that allows for an FPGA to provisioned into a secure state, allowing for secure hardware to be run in an FPGA. We then combine these two systems to implement the user space hardware provided by Cloud RTR on the secure platform provided by SDSHW, which provides our vision of user space secure hardware.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this dissertation, we propose the creation of a new application resource: user space secure hardware. User space secure hardware allows software to include custom hardware modules that are executed with the same properties as hardware implemented in silicon. We define these properties as *fixed functionality* and *fixed isolation*, meaning that any such hardware module is fixed to provide only the functionality that it was designed to perform without interference or observation. These properties are the fundamental motivation for implementing secure systems in hardware, derived from the fact that hardware systems are "fixed" when they are manufactured, meaning that their functionality cannot be changed. By exposing these properties to user space programs, hardware implementations can be provided directly by software to provide custom security features, and these features will still have the same protections as if they were manufactured directly in silicon.

Secure Hardware Advantages and Disadvantages

We need secure hardware because many security applications can only be implemented with the properties provided by a silicon implementation. This is because these isolation and functionality properties cannot be provided completely by software, as the functionality of software can, by definition, be changed. Once manufactured, the functionality of silicon cannot be altered. As such, this functionality can be implemented to provide a security primitive, such as secure

key storage or isolated computation. Because the only way to access these features is through the API (application programming interface) that the hardware exposes, the security the hardware provides is enforced by the immutability of the hardware; this API cannot be changed since it is implemented as silicon. These properties have been used to implement various secure systems, including brute-force resistant user data encryption [1], authenticated data feed systems [2], scalable blockchain transactions [3], and have the promise to address many of the security challenges of cloud computing [4].

Use of hardware systems does not come without problems, though. Hardware implementations are as prone to implementation flaws as software, but due to the immutability of a hardware implementation, these flaws are much more difficult to fix. More problematic is the fact that there are few different implementations of secure hardware systems, so the set of hardware features of one platform is not likely to be supported by another. Therefore, it is likely that there is not a single platform that supports all of the hardware features an application needs, and if a flaw is discovered in an existing platform, moving to another means sacrificing functionality.

Problems Caused by Inflexible Feature Sets

This lack of feature sets is a problem for application development, because it locks developers to both a certain silicon manufacturer and certain system implementations from that manufacturer. There are some cases where flaws can be addressed or new features enabled in released devices, such as by changing system firmware or CPU microcode, but developers rely on manufacturers to make these changes. However, there remains a fundamental limit to these systems: the actual functionality of the silicon cannot be changed. This leaves the developer the choice of either accepting flaws until new hardware is released or looking for a different platform that may not exist.

Our Vision of User Space Secure Hardware

For these reasons we propose user space secure hardware. We ask the question: what if applications could implement their own hardware? Having such a capability would put the design

decisions of which features to implement in hardware into the hands of developers. Our vision is for applications to be able to include their own hardware that can be launched with the same properties as physical secure hardware, essentially making secure hardware into a user space resource.

Leverage FPGA Technologies

Our vision requires that the hardware can be reprogrammed, meaning that new hardware could be loaded without the manufacture of new silicon. Fortunately, technology exists that provides this capability in the form of Field Programmable Gate Arrays (FPGAs), which are systems that implement reprogrammable hardware. Importantly, FPGAs are no longer special purpose devices that only a few can access. Reprogrammable hardware (*i.e.,* FPGAs) is starting to become pervasive in computing platforms. For example, Amazon AWS offers instances which allow developers to program FPGAs [5]. FPGAs also have a promising future in Microsoft data centers [6, 7, 8, 9], and in embedded systems such as self-driving cars [10] and mobile phones [11, 12]. In general, using FPGAs is desirable for certain classes of applications, as FPGA implementations of these applications can achieve near-silicon like performance and high degrees of parallelization, and are often used for machine learning.

Challenges of using FPGAs for Arbitrary User Space Hardware

FPGAs are more difficult to integrate as a software resource however, even in these new systems-on-chip (SoCs). The main technical challenges for using them with applications is that the FPGA is much more constrained when compared to other resources that an application has access to, such as computation, memory, storage, or networking. This means that sharing the limited space in an FPGA between multiple applications is difficult. Furthermore, the reprogrammability of FPGAs violates the properties provided by silicon, as the functionality can be changed at any time, whereas a silicon implementation derives its security from the fact that this is not possible. Both of these challenges make using FPGAs to provide secure hardware as a user space resource difficult to achieve.

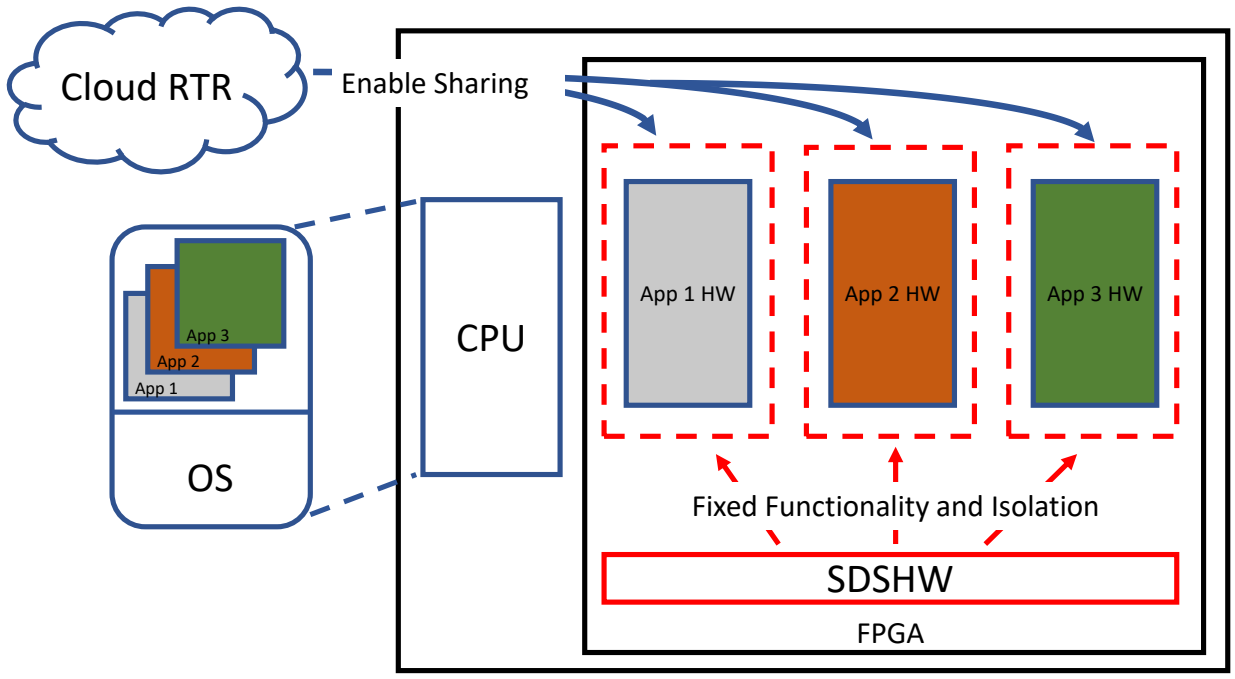

Figure 1.1: Overview of using Cloud RTR and SDSHW to allow software applications to include their own hardware, thus enabling user space secure hardware.

Overcoming FPGA Challenges

In this dissertation, we overcome these challenges and enable FPGAs to be both shareable and secure to ultimately provide our vision of user space secure hardware. We illustrate this vision in Figure 1.1, which presents how our solutions of Cloud Runtime Reconfiguration (RTR) and Software Defined Secure Hardware (SDSHW) interact.

Cloud RTR provides our solution of sharing the FPGA by compiling applications in the cloud. By leveraging the centralized deployment model of mobile applications, Cloud RTR is able to compile applications for any target platform without requiring knowledge of every applications at design time. Cloud RTR does this using existing vendor tools so that this solution can be realized today. We describe Cloud RTR in Chapter 3.

SDSHW makes the FPGA secure by having the FPGA self-provision its own security. With SDSHW, we provide a platform for arbitrary secure hardware to execute with the same properties as a silicon implementation. The self-provisioning process allows for an FPGA to have these properties. In order to still leverage the reprogrammability of the FPGA, SDSHW also provides an

update system that breaks existing trust relationships so that applications can be updated without compromising the properties provided by self-provisioning. We describe this system in Chapter 4.

We combine these in Chapter 5 to provide our complete vision by making Cloud RTR compatible with the security requirements of SDSHW. Without modification, these two systems are incompatible with each other, as SDSHW has several security requirements that break the functionality of Cloud RTR. To use them together, we move the reconfiguration system into the FPGA, allowing for Cloud RTR to reconfigure slots without compromising the security of SDSHW.

Chapter 2

Parties, Trust and Threats

In this dissertation, we present a system that allows software to use secure hardware as if it is any other shareable resource. Here, we define the exact properties of secure hardware that we are trying to provide. We do this by examining the threat model of secure hardware by showing what threats secure hardware is designed to defend against, and providing examples of applications that are or are not sensitive to these threats. Based on these examples, we distill the exact properties that secure hardware requires, present the threat model that results from these properties, and describe the trust relationships needed to enable this model.

2.1 Secure Hardware Properties

Silicon hardware is defined as a physical silicon microchip manufactured in a facility that assembles a circuit design into an array of transistors. Once manufactured, the operation of this chip cannot realistically be changed as the transistors are physically embedded in the chip; it is generally accepted that once manufactured, the layout of the transistors cannot be physically moved. This layout and interconnection of transistors forms the implementation of digital circuits that are fundamental to computing systems. These circuits provide everything from simple controller logic to sensors and convertors to complex systems such as CPUs and GPUs, and must be implemented with silicon to exist. Software is designed to be programmed to these physical systems based on

their ABIs (application binary interfaces) (*i.e.,* their instruction sets). Essentially, these hardware systems provide *features* that software can take advantage of.

When implementing a security feature in hardware, designers are attempting to take advantage of the promises that a silicon implementation provides, which are derived from the fact that any digital circuit cannot be changed. Therefore, one can implement a function that will not perform any other execution than its original design. For example, a system like a Trusted Platform Module provides an API for secure data access and cryptographic operations using secret keys. These keys are generated and stored inside the TPM and operations are requested through the API the TPM provides to the rest of the system. This API cannot be changed and the secret keys cannot be leaked out of the isolated storage maintained by the TPM because they are implemented as an isolated hardware system (with caveats, as explained later).

Essentially, secure hardware applications like TPMs rely on properties of physical hardware to ensure that the functionality of their application cannot be subverted. In comparison, software systems depend inherently on the platforms they are run on top of. This includes the physical hardware (*e.g.,* CPU), the operating system, and the various libraries that the software interacts with. This means that any of these interactions can be used to influence the operation of the software, and often times requires an application to make assumptions about these systems.

For example, a web browser password manager application must make assumptions about permissions enforced by the browser to access its internal state. Since the purpose of the application is to store passwords for a user, the application wants to only let the user have access to this data. Since web browsers read data and execute arbitrary code from untrusted sources (*e.g.,* Javascript), they must ensure that this code is properly isolated so that a malicious script cannot read the data from the password manager in place of the user. The password manager trusts the web browser to implement these controls.

More examples are communication protocols like the Secure Shell protocol (SSH), which rely on the operating system to protect import authentication data in the form of secret cryptographic keys. These keys are used by the protocol to authenticate users of the software and the operating

system is trusted to enforce permissions such that only the user that generated the keys can access them. If this function is not performed correctly, then any user or program on the system can gain access to this data and impersonate the user when interacting with the protocol. Therefore, SSH trusts the operating system to enforce file permission access control on this data, which is a feature the operating system claims to provide.

By comparing these examples of secure hardware versus secure software, we can summarize the properties of silicon that secure hardware attempts to leverage. From these examples, there is a need to trust the execution platform to operate correctly and non-maliciously, as seen by software. For secure hardware, the execution platform is the application itself. Therefore, we see that secure hardware needs two properties. The first is **fixed functionality**, meaning that the functionality of an application is difficult to modify, as a digital circuit is difficult to modify after manufacture. The second property is **fixed isolation**, meaning that the execution of an application is difficult to observe or modify, which is also provided by the difficulty of modifying a digital circuit. From the examples we have discussed, we see that secure hardware uses these properties to provide security, such as the fixed functionality and isolation of a TPM, whereas a software application such as SSH requires the operating system and libraries to provide these properties.

In the next section we discuss the various threats that these properties of silicon can be used to defend against, those that it is not useful for, and those that it is still vulnerable to. As mentioned above, silicon provides several properties based on some assumptions, but there are some advanced attacks that break these assumptions.

2.2 Secure Hardware Threat Model

As stated, security applications that are implemented in hardware implicitly rely on certain properties of silicon, which we summarize as *fixed functionality* and *fixed isolations*. These properties provide protection against certain classes of adversaries that other technologies are not able to defend against. Specifically, hardware protects against adversaries that can gain control of or

modify other parts of the system, such as the operating system or storage devices. Secure hardware also defends against adversaries that can gain physical access to systems to make these changes, which is a threat that is explicitly omitted from threat models of most secure systems. Essentially, secure hardware is used to defend against other parts of a system that may be compromised by an adversary that can physically modify the system, and can maintain this model against all but the most sophisticated attackers.

Examples of applications that need this threat model are systems that need to protect some sort of state from being observed and perform operations using this state. For example, disk encryption software is designed to protect data on a computer hard drive. The encryption is performed using a secret key that is either stored on the system itself or derived from a user's password. As such, the security of this data is maintained either by the strength of the user's password or the security of how the system stores the key. If part of the system is compromised, this secret key can be read out of a system if it is not securely protected, and a physical adversary can also simply remove the storage and perform a brute-force attack on the user's password to decrypt it.

Because of these issues, such applications often use secure hardware systems, such as TPMs, to protect this secret key. A TPM can securely store the secret key in an isolated storage system and perform cryptographic operations using it, and allows for authorization to be required before the key can be accessed. As such, an encryption system can store the secret key in the TPM and require data to be decrypted using this key, which makes decrypting the data impossible outside of the system. Furthermore, a user's password can be used to unlock the TPM rather than deriving an encryption key, and combined with rate-limiting enforced by the TPM, can prevent brute-forcing of the password. In short, using a TPM makes physical attacks against a encryption software more difficult, essentially requiring an adversary to perform an online attack that compromises the system so as to snoop the user's password in some way, such as by compromising the software itself with a keylogger, rather than being able to brute-force the user's password offline.

Other applications use secure hardware to protect other types of state. Trusted Execution Environments (TEEs), such as ARM TrustZone and Intel's Software Guard Extensions (SGX), isolate

the execution of software to prevent observation and interference with its execution and secret data and are designed to prevent even the most privileged code (*e.g.,* the operating system) from interfering with the software's operation. Secure and trusted boot systems prevent unauthorized software from executing on systems, and are meant to defend against the trusted operating system being overwritten by an adversary that has access to the physical storage of the device. Remote attestation systems prove that software is executing in a system to an external party and is meant to defend against an adversary that can force imposter software to be launched by either compromising the operating system or physically replacing the software. All these systems rely on the properties of hardware to ensure that these functions cannot be overridden, as they are specifically designed to defend against the rest of the system. The properties of hardware are needed in this case because the fundamental trust model required by software, which requires all of the parts of the system to work together, cannot be relied on in the face of adversaries that can manipulate the rest of the system.

Secure hardware is also vulnerable to a class of advanced physical adversaries that have the resources to perform more invasive and expensive attacks. These attacks involve circuit reverse engineering, chemical analysis, and chip deconstruction that requires highly sophisticated equipment and facilities, and is quite destructive. These 'decapping' techniques, though unreliable, theoretically allow the state of a digital circuit to be inspected post-manufacturer, which can lead to secret data being retrieved from a protected storage system. These techniques often involve the removal of hardware from a system and generally result in the destruction of the physical chip, and so are one-time attempts. However, these attacks are more difficult to perform and are expensive. Manufacturers of secure hardware can chose to invest more to prevent these attacks, but there is a tradeoff between security and implementation cost for these solutions. Secure hardware therefore can be vulnerable to these threats, but it also forces adversaries to use these more expensive techniques to attack the system.

Finally, secure hardware has several threat vectors related to the physical requirement for silicon to be powered. Differential power analysis (DPA) is a technique where the power consumption

of a system is analyzed during its execution to determine its state, which otherwise cannot be observed. This type of attack is theoretically possible against all hardware systems, since they use power, but the difficulty depends on the power consumption of the application under attack and how much this changes during execution. It is possible to design hardware and applications to be resistant to this attack, but comes with their own tradeoffs (*e.g.,* higher power consumption to mask usage changes). Additionally, hardware systems can be disabled or have their execution interrupted if an adversary has control of the power supply to the system. The adversary can therefore perform a denial-of-service (DoS) attack against any hardware device if they have this capability, but it is difficult for this to be targeted to a single hardware application in a complete system. These power vulnerabilities, however, are applicable to all systems implemented in silicon.

We note that implementation flaws of silicon secure hardware is out of scope of this threat model. Applications are still trusted to be correct; silicon does not make an application any easier or safer to design (in fact, the opposite is often true). Silicon manufacturers have introduced a number of systems to mitigate the risk of an incorrect implementation, as any silicon system that has a flaw cannot be patched due its immutability. These mitigations therefore attempt to define as much of a system's functionality in software, such as through CPU microcode or firmware, but cannot completely solve the issue. The system we propose in user space hardware does allow for a developer to easily update the functionality of the secure hardware they design by leveraging the flexibility of an FPGA, but does not prevent an incorrect implementation from being created, just as traditional secure hardware does not provide this protection.

In this dissertation, we present a system that makes secure hardware into a software resource, meaning that it can be accessed and programmed from user space. We do not add any new protections or expand the threat model of secure hardware, but we do design our system so that this user space secure hardware maintains the same threat model as existing secure hardware, with all of its benefits and vulnerabilities. However, because we allow secure hardware to be used as a software resource, we introduce a problem resulting from the fact that an FPGA needs to be shared between applications. We discuss this in further detail in later chapters, but in summary, we introduce a

new threat vector resulting from the fact that secure hardware can be loaded and unloaded after a device has been manufactured, meaning that the property of fixed functionality may not always hold. In order to maintain the same threat model as secure hardware, we need to also address this new threat vector.

In Chapter 3, we show how to enable user space hardware by sharing an FPGA, but that system makes no claims about the security of this hardware, and in fact this new threat vector is introduced. In Chapter 4, we show how hardware in an FPGA can be executed with the properties we have identified by addressing this new threat vector, and in Chapter 5, we show how this threat model can be expanded to user space hardware so that sharing of secure hardware between applications can be enabled. In the next sections, we define the exact parties involved in creating and using secure hardware and the resulting trust relationships that must be maintained in order for the threat model we have presented to hold.

2.3 Definition of Roles

The threat model for secure hardware relies on a number of parties that must perform certain actions for the threat model of secure hardware to be possible. For example, secure hardware silicon is manufactured by one party, which must be trusted to implement features correctly and not to introduce backdoors or vulnerabilities into the system. In this section, we identify the roles that are required to provide and use secure hardware today. In the next section we discuss the trust model of secure hardware, which describes the relationship between these roles and how our system changes these relationships.

We have identified six fundamental roles that are involved in the manufacture, design, and use of secure hardware systems, both in existing models and the systems presented in this dissertation. These roles are intuitive, but often multiple roles can be assumed by the same party (*e.g.,* a person or organization) and so do not always appear to be separate in the real world. For this reason, we define these roles here explicitly and provide examples of how they are assumed.

We define these roles as: the silicon manufacturer, the hardware assembler, the system provisioner, the application developer, the application distributor, and the end user. Each of these roles is assumed by some party in any device, but many are assumed by a single party, in different combinations for different devices.

- **Silicon Manufacturers** manufacturer a physical chip, such as a CPU, a GPU, or other integrated circuits. This party maintains its own manufacturing and design facilities and its own supply chain, and sells the manufactured systems to hardware assemblers. Examples of this role are chip manufacturers, such as Intel, AMD, or Qualcomm.

- **Hardware Assemblers** are the party that combines a number of different physical components into a single device, such as a server, laptop or desktop computer, or a mobile device. These systems are then provided for sale to customers. Examples of this role are system manufacturers such as Dell, Apple, or Samsung.

- **System Provisioners** are the party that determines what software (and/or reprogrammable hardware modules) run initially or can be run on a device. As such, they provision the system with an initial configuration and determine how (or if) updates can be made to this configuration, or if the system can be reconfigured by another system provisioner in the future. In the case of personal computers, this can be done by the end user, but is often done by system distributors who manufacture devices in the role of the hardware assembler. Google is an example of a provisioner that does not assemble the system, as Google contracts with a hardware assembler such as HTC or LG to assemble their smartphones.

- **Application Developers** implement the software that is run in a device, including the initial configuration chosen by the provisioner. For example, developers such as Microsoft implement operating systems, whereas developers such as Mozilla, Dropbox, or Oracle provide individual applications.

- **Application Distributors** provide applications to devices. Often, this application distribution system is integrated into the operating system (*e.g.,* the Google Play store, Microsoft

Store, Linux package manager) and is the only way of providing new applications (*e.g.,* iOS App store, video game consoles). Most of these distributors are only applicable to a single platform, and so are maintained by the providers of a device's operating system, which is selected by the system provisioner, but some, such as the Linux package management systems, can be shared between different platforms.

- **End Users** are the individuals that use the device after it is configured, such as after a provisioner has purchased and provisioned a device and offered it for sale to a consumer. Enterprises are an example of an end user that would provision or re-provision a device, as a company's IT department may choose to replace or add to the software running on employees' computers.

We will use these role definitions throughout the rest of this dissertation in order to explain how different systems interact. In the following chapters, we will show how the responsibilities of these roles introduce problems and how they can be changed to provide our solution. In the next section, we show the trust relationships between these roles that are needed for the threat model of secure hardware to be realized.

2.4 Secure Hardware Trust Model

The subject of trust is another important definition for any secure system. Any secure function should define which entities are trusted to perform what actions, and how much that trust can impact the system's security if violated. We have already presented the threat model for secure hardware. In this section, we define which parties need to perform what actions in order for this threat model to be realized.

As the concept of trust is ambiguously defined in security research, we define trust as the acceptance of any claim about a system made by an involved party, specifically a claim about a security property. For example, an application developer *trusts* a silicon manufacturer to provide a

feature, who then claims that the feature is implemented correctly and that there are no backdoors or other ways to exploit it.

For applications with no expectation of security, this trust relationship is simpler, as a user can verify the operation of the application to prove that the system performs correctly and any other features the manufacturer implemented are superfluous if they do not impact the execution of the hardware. However, for secure hardware, these extra features do matter as they can potentially compromise security, such as a backdoor to leak private data in a system designed to protect it. If these features are undocumented, they are difficult to discover due to the inherent introspection resistance of hardware, which results from its fixed isolation property. A user must instead accept the promises a silicon manufacturer makes based on other criteria, such as the reputation of the manufacturer or the risk of the actual data that would be compromised. Any such security system that relies on such a trust model therefore is vulnerable to adversaries that can subvert it. Different examples of trust models for secure hardware are discussed further in Chapter 4.

The six parties that we identified previously in this chapter have different impacts on the trust of a secure hardware application. For existing hardware implementations, each of these parties makes or must accept certain claims from others. The trust chain for existing secure hardware and user space secure hardware is shown in Figure 2.1. This figure shows two type of trust relations: functional trust and process trust. Functional trust is the trust of a party to design and build a system, but is only required at design and manufacture time. Process trust is the trust of a party to maintain a service or business process for the lifetime of a device for it to provide its function, such as the protection of cryptographic keys. Examples of these process relationships in existing systems can be seen in SGX and UEFI secure boot. SGX requires the use of Intel's services in order to create new enclaves or perform remote attestation, but this relationship can be potentially abused maliciously or for revenue generation [13]. UEFI secure boot relies on the protection of a set of secret signing keys, but the security of these keys was compromised by Microsoft [14, 15, 16]. In both these systems, a process trust relationship exists that can be used to compromise the system with a flaw existing in the hardware's actual implementation.

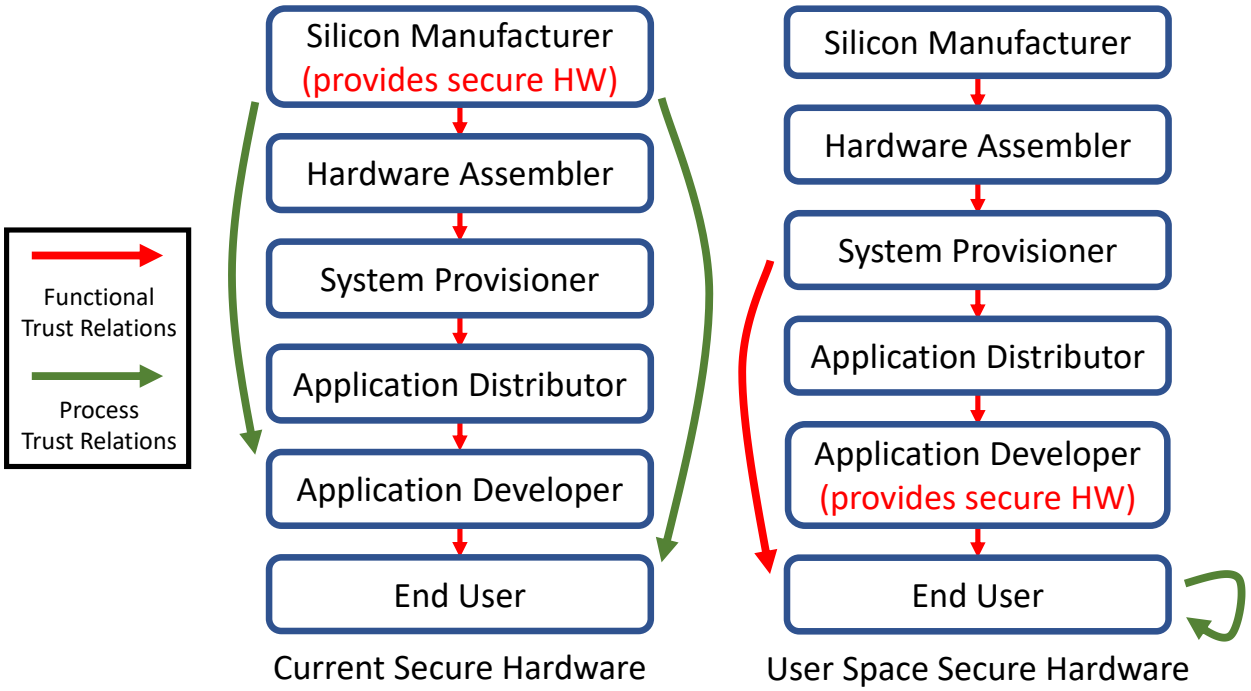

Figure 2.1: The trust relationships between existing secure hardware (left) and user space secure hardware (right). Red arrows indicate trust for the implementation of functionality, whereas green arrows indicate trust for a process that the party depends upon. The current model requires trust in processes as well as implementation of functionality for secure hardware to be secure. User space secure hardware changes this model to replace this process trust of the silicon manufacturer to a one-time functionality trust of the system provisioner. The device and the end user are the only ones responsible for maintaining the security of the secure hardware in this model.

These process trust relationships also exist in FPGAs, as is discussed further in Chapter 4. The problem with an FPGA is that the functionality trust is not one-time, as it is with secure hardware. In Chapter 4, we present systems that change this to make the functionality trust one-time, just as in silicon secure hardware. To do this, we need to break the process trust model used by FPGAs, which results in the second column in Figure 2.1. In this model, we require the system provisioner to put the FPGA into a single secure state that cannot be changed, and require that only this secure initial state can authorize future states. This removes the existing process trust models of secure hardware, as only the secure hardware can affect its own state, but also requires an additional functional trust relationship with the provisioner.

Chapter 3

Cloud RTR: Enabling User Space Hardware

Enabling end-users to install applications of their own choice has been a cornerstone of each wave of computing. From vertically integrated mainframes transitioning to the personal computer with complete user control of software, and from early vertically integrated personal devices (phones and PDAs) with manufacturer-packaged applications transitioning to modern devices with app stores, this trend has been continually repeated. We propose a similar model that we call user space hardware, which allows systems to present hardware as a software resource to applications. This is a first step toward our vision of secure user space hardware. To provide this, we must overcome a fundamental challenge for any system that supports multiple applications and users: **how to share limited resources?** In this case, the question is how to share limited FPGA logic amongst many applications.

3.1 Introduction

User space hardware is needed because smart phones do not always provide the hardware features that application developers need, and in other cases provide features that are never used. This is because silicon manufacturers and hardware assemblers designing smart phones must operate

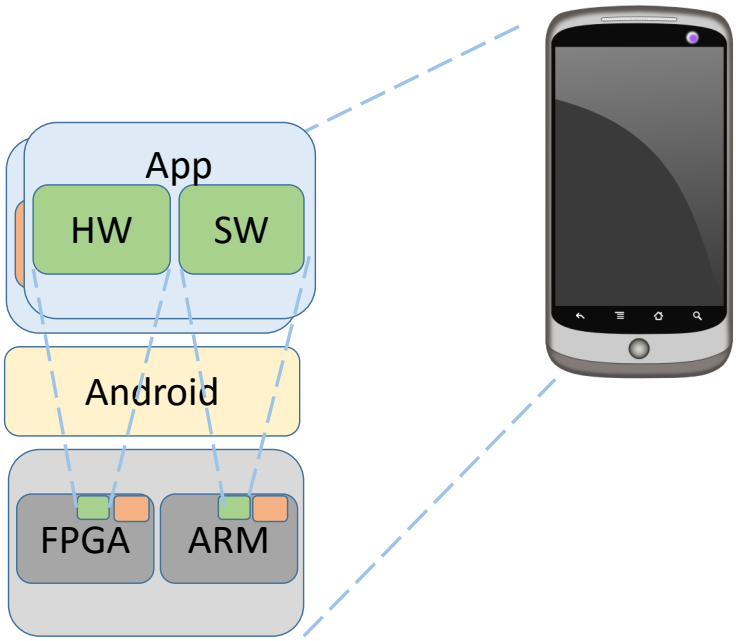

Figure 3.1: Smart phone with a processor (ARM) coupled with programmable hardware (FPGA).

under a number of constraints – form factor, functionality, cost, energy use, etc. This leads to the assembler making a number of decisions regarding the various tradeoffs. These decisions, however, lead to the case where the device has both too little (the application developers/end users want more) and too much (the application developers/end users don't use what is there). *What if there was a way to put these trade-offs into the hands of end users and application developers?*

In this chapter, we present our vision for 'user space hardware' (illustrated in Figure 3.1). This vision incorporates programmable hardware, such as an FPGA, into a smart phone[1], and extends a mobile operating system to allow for application control of the current hardware configuration (*e.g.,* by including the hardware configuration with their app). The high-level idea is to couple software-like (re)programmability with hardware-like performance and immutability. In providing programmability, hardware assemblers empower application developers (and by extension end users) with the ability to influence these design decisions. As such, we can realize part of our

[1]We envision this as being commercially available smart phones, not just in prototyping devices – a vision supported by the commercial availability of system-on-chip devices which already couple an ARM processor that is widely used in smart phones (such as the ARM Cortex A9 processor found in the iPhone 4) with reconfigurable logic [17, 11], with or more recently the ARM Cortex A53 [12], and further supported by recent advances by vendors where hardware modules can be designed using a high-level language, such as C++ [18].

overall goal of user space secure hardware by first making hardware into a software resource, *i.e.,* user space hardware.

Application developers, for example, would be able to introduce (and deploy) new communication technologies, such as those that work on the emerging dynamic spectrum access paradigm [19], where they can perform 'software' radio at the needed hardware performance levels and gain system wide benefits (*e.g.,* from not needing phones to include many dedicated radio interfaces). Developers will also be able to introduce new accelerators, such as for cryptography or other parallel processing that improve overall performance and efficiency. Finally, developers will be able to introduce independent co-processors, which can, for example, provide additional security capabilities [20] not possible in today's smart phones. In general, we introduce programmability of the smart phone hardware by creating an architecture centered on an FPGA with an embedded processor – with which, as we've seen with other programmable technology, such as GPUs and FPGAs in other contexts within the network systems community, developers will find creative ways to use the available processing power [21, 22, 23, 24].

Previous research [25, 26, 27] has proposed adding reconfigurability to mobile devices, but they have limitations that prevent them from being used today, such as lacking a method to share FPGA hardware, a distribution system for applications, or integration into modern operating systems or devices. Therefore, there are a number of challenges that need to be addressed to make this vision possible. First, we need to be able to share an FPGA between different smart phone applications – existing FPGA hardware and software are heavily centered on running a single application and not on an idea of temporal or spacial sharing of resources. Second, we need a way to distribute apps and their hardware modules to the correct platforms; hardware modules are not easily relocatable between different platforms. Finally, we need the ability to manage the FPGA so that applications only have access to authorized resources; while processors have been adapted overtime to isolate running tasks, FPGAs have not.

In this chapter we present our Cloud RTR system that addresses these challenges. We introduce a system-level contribution that makes use of cloud technologies and builds on existing FPGA

technology that together solve a problem that has eluded researchers for years. Specifically, we make the following contributions:

A slot-based solution that allows for practical FPGA sharing: A central need to be able to allow apps to span software and the FPGA hardware is to enable the FPGA to be shared, as apps will be running concurrently. This approach is based on runtime reconfiguration (RTR), or the ability to change an FPGA's configuration at runtime. Specifically, the Cloud RTR system builds on the idea of "slots" [28, 29, 30], or areas of the FPGA that can be reconfigured separately and shared between applications. To make this practical where previous systems have failed, it provides a new approach to slot-based reconfiguration using a compilation system that abstracts away the underlying FPGA requirements. The resulting platform supports the use of slots at runtime, whereas previous systems only support slots at design time, and can share the FPGA between multiple parties, as we discuss next. Further, it introduces operating system services to manage slots at runtime to allow for on-demand access from apps.

An app store-based approach that allows for multiple parties to distribute apps: Without operating system and binary compatibility, envisioning a system which allows for multiple parties to create apps and have them be distributed to a wide variety of devices may seem difficult. We introduce a new distribution system which extends existing smart phone application distributors (*i.e.,* app stores) to allow for both the compilation and the distribution of apps with hardware. Developers can upload apps with hardware to an extended app store, which will interface with the compilation system in order to generate the required slot configurations. We extend the app store system further to ensure that these configurations are distributed to the correct devices in packaged apps, and we provide corresponding operating system support in order to install them.

In addition to the above system-level advances which enable the apps with hardware vision, we make the following contributions which evaluate and demonstrate their use:

Evaluation of the computational requirements of Cloud RTR: While our approach of performing some compilation in the cloud is, to a degree, simplistic, we feel that the fact that it has not

been done before does points to its novelty. Importantly, we go beyond simply proposing to compile in the cloud and extend our work to fully evaluate the computation requirements of such an app store to support this using data about the current app market ecosystem. We show that compilation throughput per machine ranges from 51 to 121 apps per day, which translates to needing 981 servers to support an app ecosystem where 1% of all apps use the reconfigurable logic for a case where there are 1000 phone variants.

Demonstration and evaluation of three applications: In Section 3.2, we describe three example categories of applications that will benefit from user space hardware, independant of the overall goal of user space secure hardware. For each, we implemented and evaluated a representative application (Section 3.6). Our evaluation of an app which offloads to a hardware based Quadrature Amplitude Modulation (QAM) module (a representative software-defined radio application) shows a 40x speedup and a hardware-based AES module (a representative cryptography application) shows a 3x speedup (including all of the interface between hardware and software). Additionally, our evaluation of a simple memory security scanner (a representative architectural enhancement) that is capable of searching the entire system address space only results in 3% overhead for other software running. Finally, to understand the considerations when integrating into existing and complex applications, we modified the open source and widely used Orbot [31] Tor [32] client for Android to include and use a hardware cryptography module (Section 3.7).

3.2 Motivation (Why an FPGA)

The premise of incorporating an FPGA into a smart phone lies in the general benefits of an FPGA – that it provides hardware-level programmability which will enable phone manufacturers to defer some decisions about tradeoffs to the end user and enable developers with the ability to innovate in the hardware space.

Here we discuss a few examples that help motivate an FPGA within a smart phone, including a description of a demonstration application that we implemented for each of these categories.

3.2.1 Architecture enhancements

For the first set of motivating examples, we present several architectural enhancements that have been proposed in the research community that each required a hardware plug-in and were targeted at a server. With our work, similar benefits could be brought to a smart phone.

CoPilot: CoPilot [20] is a PCI card designed to detect rootkits. As rootkits execute at the highest privilege, detection mechanisms at the same (or lower) privilege are presented with a significant challenge. The CoPilot PCI card is independent of the processor and operating system and has access to all memory via the PCI bus. Rootkit detection (or more generally, security applications) have tremendous potential with the introduction of an FPGA within a smart phone.

Somniloquy: The Somniloquy [33] work observed that the energy consumption on servers was impacted by a number of low-rate types of tasks that prevented the servers from entering the power saving states. As such, they proposed a small, low-power processor that could perform these tasks, and if needed, trigger the main processor to exit a low-power state. In the case of a smart phone with an FPGA, similar types of activity have been observed in smart phones [34], so a small co-processor in the FPGA fabric could provide a solution (while also enabling the main processor to shut off completely). We leave full exploration of power as future work.

As a demonstration of architectural enhancements, we have implemented a memory scanner module as a simplified proxy for a CoPilot-like function, that scans our device's system address space.

3.2.2 Software-defined Radio

A great deal of research has resulted in many innovations in wireless communications, which allow wireless interfaces to have better performance or more functionality. Research papers in this space commonly use FPGA platforms (such as the WARP Board [35, 36]), or devices to interface to high performance desktop machines (such as the USRP [37]) in order to meet the needs of the new innovation. While these papers provided promising research results, there is little opportunity

for deployment – requiring the researchers to commercialize the technology, or get adoption from a major chip vendor.

With a smart phone that has an FPGA along with a more flexible radio front end (*e.g.,* a tunable antenna), developers of a new communication protocols could simply create an app, enhancing the impact of the research. This architecture also has benefits for production systems, as existing devices could be upgraded to new wireless systems without requiring replacement, such as upgrading such a system from 3G to 4G wireless technology.

As a demonstration of an SDR application, we have included an example implementation of a Carrier Phase Recovery Loop for a single carrier Quadrature Amplitude Modulation (QAM) demodulator. QAM is a representative building block in signal processing applications including many real-world modulation systems.

3.2.3 Cryptographic and Parallel Processing

FPGAs have the ability to perform large amounts of processing in parallel. This allows them to achieve higher throughputs and lower latencies.

An exemplary application for FPGA acceleration on a smart phone is cryptographic processing, as it both faster in an FPGA and widely used – including the encryption of internet communication using SSL and communication protocols such as Tor [32], the accountable internet protocols [38], and Named-data Networking [39] For example, an FPGA (Altera Stratix V) was shown to be 520 times faster than a general purpose processor (Intel Xeon E5503) for AES encryption (and 15x speedup over an AMD Ratheon HD 7970 GPU) [40]. While the exact numbers will depend on a number of factors, this is illustrative of the potential.

Parallel processing goes beyond cryptography. One recent example used an FPGA based server [41] to implement common functions used in analytics (search, fuzzy search, and term frequency), and in each case demonstrated that it would require 100-200 servers running Spark [42] to match the performance. This example is geared towards cloud scale applications, but we believe this would allow us to perform some analytic processing locally.

As an example of this type of application, we have implemented a 128-bit AES encryption module that can encrypt an arbitrary number of 128-bit contiguous regions of memory. We also incorporated this AES module into the Orbot Tor client (Section 3.7).

3.3 Past Attempts

A central challenge in reaching our vision relates to how to share the FPGA between applications and the system. That is, we wish for multiple apps to be able to simultaneously use some of the FPGA's programmable fabric, while at the same time allowing the operating system to use some of the programmable fabric as well (*e.g.,* to connect to some I/O devices).

The core concept required is runtime reconfiguration, or the ability to dynamically change the FPGA's configuration (completely or partially) at runtime while it is still operating. Despite over a decade of research in runtime reconfiguration [43, 44, 45, 46, 47, 48, 49, 50] there has yet to be a practical solution which would enable hardware modules from various sources to be loaded onto a variety of platforms.

3.3.1 Why is sharing an FPGA difficult?

The main challenge in achieving FPGA sharing is ensuring that the apps' modules in the FPGA do not conflict with each other, or with other logic that is present in the FPGA. FPGAs are difficult to share because a complex mapping of resources must occur in order to generate a configuration for an FPGA. This is because application's logic must be mapped to physical resources in the FPGA, and connections must be made between these locations, just as in any physical circuit.

As an example, consider Figure 3.2, which illustrates a single module to be loaded into an FPGA at runtime. The dotted area indicates one possible location to put that module. As indicated, however, there will be contention for resources – *i.e.,* this module cannot co-exist with the current FPGA configuration. Because of this, the partial reconfiguration mechanism supported by the vendors (Altera and Xilinx) comes with great restrictions – the modules can only work with a

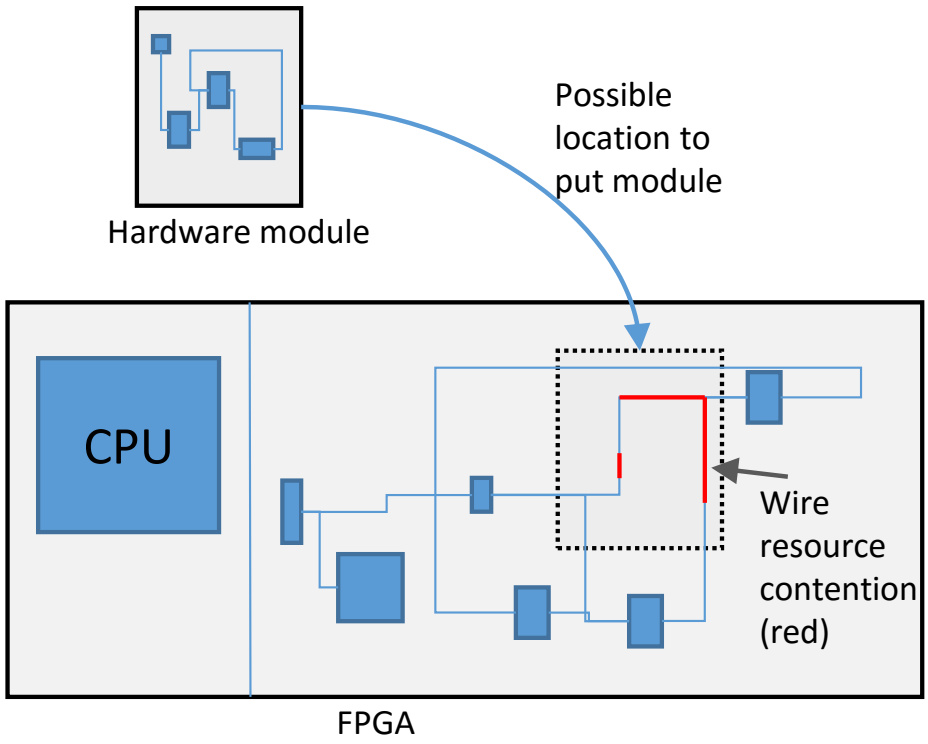

Figure 3.2: Example of partial reconfiguration in a running FPGA configuration.

single design (in our case, they wouldn't work across phone architectures), and they can only be loaded into a single location. These restrictions make partial reconfiguration unusable in its current form to enable apps with hardware.

More general runtime reconfiguration approaches have been proposed in the research community that fall into one of two categories, which we describe next. In general neither of these approaches is practical.

3.3.2 Soln. 1: runtime Place and Route

The first approach is to perform place and route at runtime [51, 52, 53] (rather than when it is normally performed – at design time). As background, place and route is a computationally expensive, NP-complete task of first mapping logic elements from the design (place) and then determining a collection of wire resources to use to connect the logic elements from the design (route).

This approach enables reconfigurable modules to be created entirely separately from the FPGA configuration. They can be loaded into the FPGA by being placed around existing hardware and connected with free wiring resources.

This is a general approach and supports our model, but there are two major problems. First, place and route can take a long time, depending on both the size of the reconfigurable module as well as the sparseness of the current FPGA configuration – *i.e.,* if there are few resources available, it will be a more difficult task to find a solution. The implication relates to the second problem – that a solution is not always possible, which means that the app would fail to load.

3.3.3 Soln. 2: Slot-based Reconfiguration

The second method that has been proposed also seeks to support a general approach where the static design and the reconfigurable modules can be created independently. This approach does so by reserving empty and identical areas in the static design [28] [29] [30]. These areas, or slots, are analogous to PCI slots on a motherboard, where any card can be plugged in independent of the processor. In this case, the 'cards' are partial bitstreams (a binary file used to configure an FPGA). Two constraints emerge:

Partial bitstreams need to be relocatable – So that a partial bitstream can be loaded into any slot, each area needs to be identical. This is not difficult from a logic standpoint, as FPGAs are fairly regular structures. In order for the static and reconfigurable portions to be able to communicate, however, there need to be wires that cross the boundaries, which in turn, need to be identical for each slot. This puts incredible strain on the creation of the static design, to the point of not being practical (because place and route becomes very constrained if certain circuit elements need to use certain physical wires).

Partial bitstreams cannot conflict with the static design – That is, when loading a partial bistream, it cannot, for example, use a wire that the main system design used (and vice versa). To achieve this, the static design is highly constrained to reserve areas such that no logic is used (generally, easy to achieve) and such that no wires are used (in Figure 3.2, this would mean that

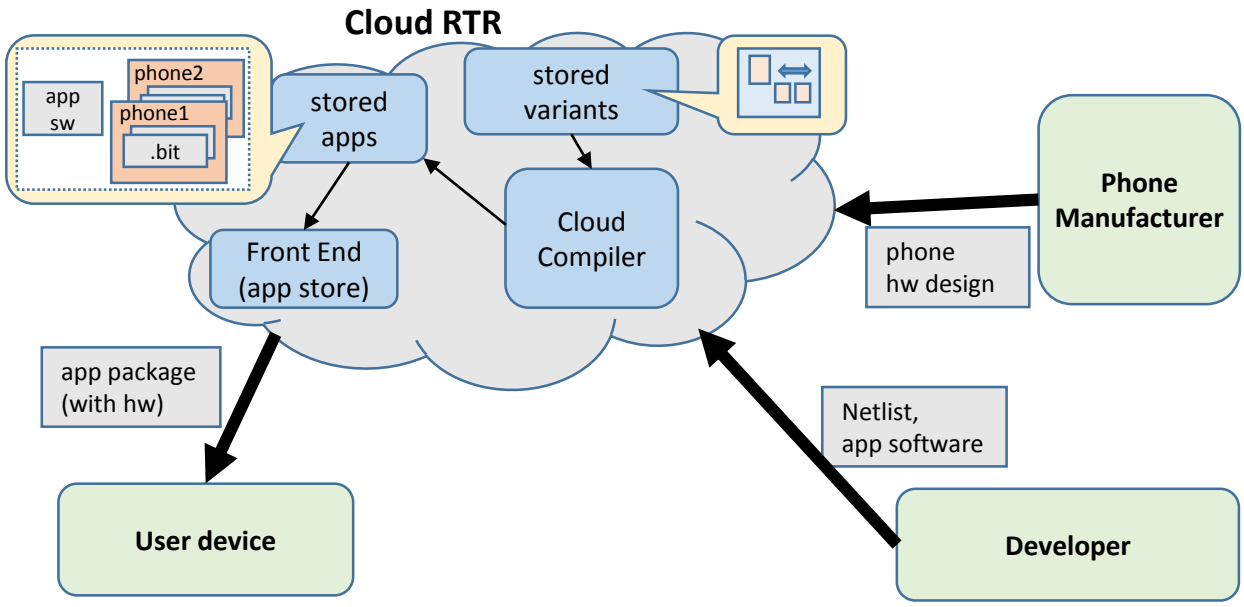

Figure 3.3: Cloud RTR approach to the generation and deployment of apps with hardware

static portion of the design would not have been allowed the wires that are in the dotted area). Such constraints are ultimately possible (through a painstaking process of reverse engineering and over-constraining), but highly constrain the static portion of the design – forcing wires to be routed around these areas, causing them to be extra long and resulting in congested areas.

In short, this is a good abstraction, but not practical.

3.4 Cloud RTR: A Practical Approach For Sharing the FPGA

In order to realize the user space hardware vision, we need two things. First, we need a mechanism to be able to share the FPGA resources – *i.e.,* a practical runtime reconfiguration mechanism that overcomes the limitations of past solutions in terms of usability and deployability. Second, we need a mechanism to be able to manage the apps at runtime. Here, we describe our novel solution for enabling FPGA sharing, and in Section 3.5 we describe our system support for runtime management of apps.

3.4.1 High-level Overview

Central to this design, we adopt the general idea of slots – that is, reserved areas within the FPGA where modules can be loaded. As previously mentioned, we believe this is a good abstraction, but the previous realizations of it are not practical. The key difference with our approach is that slots are less constrained – only logic resources need to be left free (which is easier), but the wiring resources within these areas can be used by the static design logic (*i.e.,* the portion of the FPGA configuration that does not change and provides system functionality for different phones). Other key differences with this approach are that the reconfigurable modules can (i) work with multiple slot sizes, (ii) work with multiple slot signaling interfaces, and (iii) be targeted at various end-systems.

The key idea to enable this is that by leveraging the delivery model of mobile apps (*i.e.,* via an app store), we can effectively merge the modules into various static designs in the cloud, before delivery to the end user. We call this Cloud RTR (RTR for runtime reconfiguration). As illustrated in Figure 3.3, each phone manufacturer and app developer would submit their design to the Cloud RTR system, and the Cloud RTR system would perform a compilation step to enable a general runtime reconfiguration mechanism.

In this section we describe the architecture of the phone in order to support this model (Section 3.4.2), how the apps are designed to work within the framework (Section 3.4.3), and finally discuss how Cloud RTR performs the compilation (Section 3.4.4).

3.4.2 Static (Phone) Design Architecture

The key requirement for the phone's design lies in the ability to support interfacing the reconfigurable modules with the rest of the system resources. Described below are the main components, which are also illustrated in Figure 3.4.

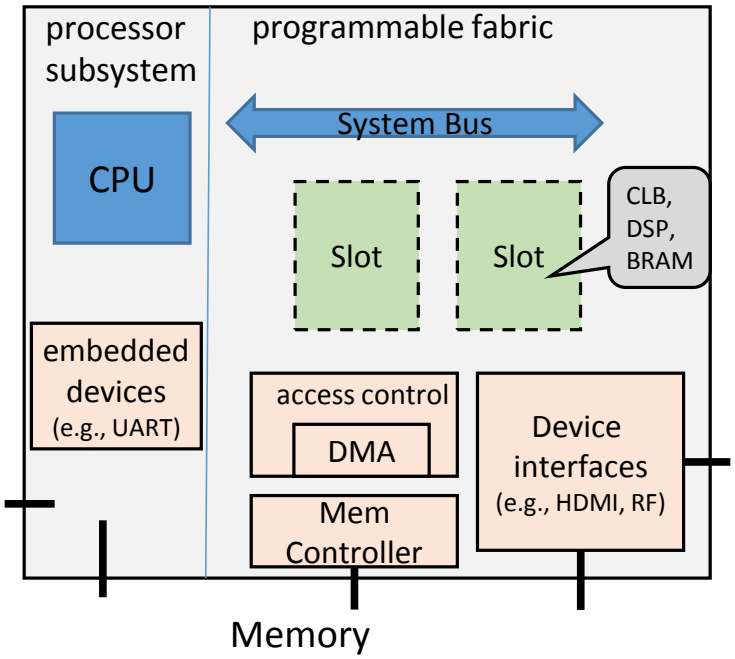

Figure 3.4: Example static FPGA design.

Slots

Slots should have enough of all types of resources to be useful. Today's FPGAs can contain (i) configurable logic blocks (CLBs), which can implement any logic function of N inputs, (ii) block random access memory (BRAM), which are small memory elements (*e.g.,* 36 Kb in the FPGA we use for implementation), and (iii) digital signal processing (DSP) blocks, which are custom building blocks geared toward signal processing applications.

Slots will also need to be able to access various system resources and expose an interface for communication with the processor. For this, we expect that all slots will allow access to (i) a system bus for communication with the processor, and (ii) a direct memory access (DMA) controller for access to system memory.

Module-to-Memory Interface

In order to provide performance benefits, the modules need to be able to directly access CPU-accessible system memory. A DMA controller that is accessible by the hardware modules would allow for modules to access system memory without involving the processor (providing the greatest performance and flexibility). To achieve this, we also need a security module which performs

access control – that is, one which limits what memory each hardware module can access and is configured by the operating system.

Processor-to-Module Interface

The ability to stream from memory will be important, but the processor also needs to be able to directly interface to each module. This interfacing is achieved through the use of, for example, a system bus (such as the ARM-based Advanced eXtensible Interface, or AXI).

Device interfacing and other misc. logic

The rest of the static design will include interfacing to the various devices that will connect to the FPGA. Some devices, such as a UART, may have interface logic included in the processor subsystem, but the rest, such as interfacing to a tunable antenna, may go through the programmable fabric with custom logic to interface with it. These devices will be connected to the general interface of the slots, allowing for manufacturers to include custom peripherals without requiring new slot definitions.

3.4.3 Reconfigurable (App) Module Architecture

In the previous slot-based approaches, the reconfigurable modules are designed for a specific slot design (device, interface, etc.). In this approach, we abstract away the ultimate target such that app developers can develop reconfigurable modules that can be loaded onto a variety of platforms. Of note, the reconfigurable modules in this approach can (i) work with multiple slot sizes, (ii) work with multiple slot signaling interfaces, and (iii) be targeted at various end-systems.

Here we describe the design of an app, with the various components illustrated in Figure 3.5.

App Hardware

The first major component is the **app hardware**. In order to match the skills of app developers, we focus on the high-level synthesis (HLS) design flow [18] that has emerged in recent years which

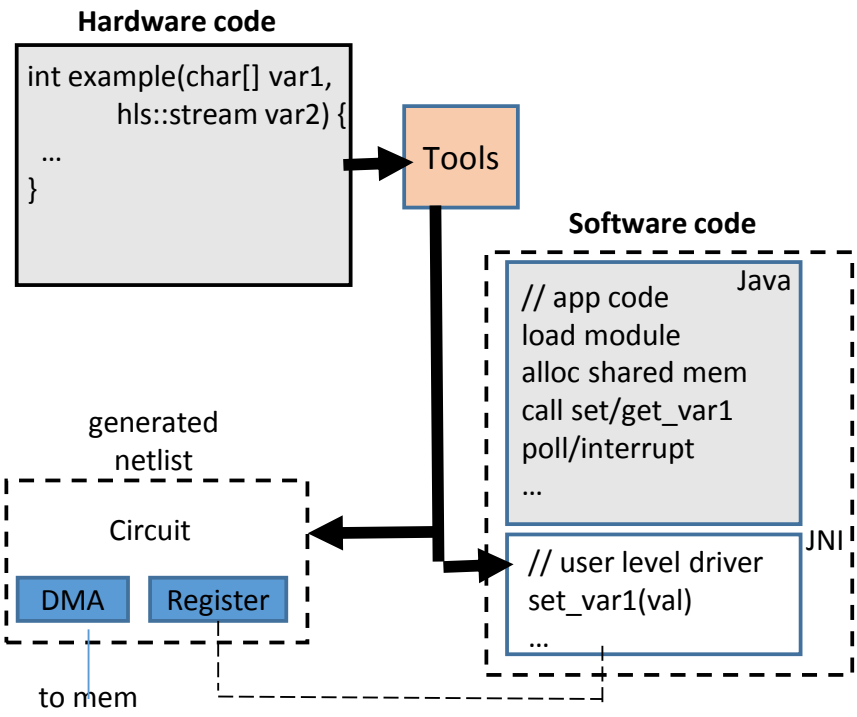

Figure 3.5: Example app design.

allows developers to use a high-level language (*e.g.,* C) to describe hardware modules[2]. *What this means is that the argument that FPGAs are hard to design for, and therefore not accessible to the software app developers, is quickly becoming invalid.*

The app hardware (in this example) is written as a C++ function, *example()*. The parameters to the function describe the interfaces to the rest of the system, such as char arrays (*e.g.,* var1), which describe memory mapped registers accessible to the processor or streaming memory interfaces (*e.g.,* var2, which has the type hls::stream), that allow for streaming data from memory (when connected to DMA hardware in the static design). This description is valid C++ code that can be compiled and tested as software, which can simplify hardware testing.

While there will need to be some consideration by developers, in general developers will not need to be fully aware of the hardware architecture. For example, the exact bus signals for communicating with the module are not directly used, but are instead inferred based on the types on the

[2]The developer can use a hardware description language, but will then need to manually provide the interfacing hardware and software, which are automatically created with high-level synthesis.

function parameters (such as how to perform data transmission handshakes or send valid signals). With this, the same module could actually target various hardware interfaces (*e.g.,* if different handshake protocols or signals are used, or if different bus widths are available). Developers do need to consider the size (resource utilization) of their hardware modules to ensure they will fit in a particular slot size. We envision standard slot sizes will emerge (much like screen sizes), and in this design flow we allow for modules designed for one slot size to always be instantiated in a bigger slot.

App Software

The **app software** that the developer writes will be mostly the same as current apps (*e.g.,* written in Java for Android apps). The only difference is the loading of and interfacing with the hardware module. To load, the app will submit a request to a system service to load the bitstream (*e.g.,* via an intent in Android).

To interface with the module, the app will use the functions in the user-level driver generated by the FPGA vendor's high-level synthesis tool when synthesizing the design (the process which generates the FPGA hardware from the C++ code). This driver is low-level code that runs within the same process as the application and provides functions that can be used to interface with the reconfigurable module. Functionality includes mapping memory regions (*e.g.,* via *mmap()*) that both the reconfigurable module and the app will access. It also provides functions to access the various registers (the char[] variable) through functions like *set_var1()*.

3.4.4 Cloud Compiler (in the App Store)

The Cloud RTR compiler is responsible for ensuring that an app's hardware module(s) can be loaded into a variety of target devices (smart phones). Rather than working around the limitations of the vendor tools, we work within their constraints, resulting in a practical solution. Recall that the vendor tools have a partial reconfiguration design flow which has the constraints that a module can only be used for a specific static design and target FPGA and for a specific location within that

static design. Working within that, the Cloud RTR compiler will simply use the vendor tools to compile the module for every static design variant and for every possible slot within each variant.

The end result is a data structure stored within the app store that looks like the following (where a.bit...e.bit are individual partial bitstreams):

```
[phone 1:
  [slot1:a.bit, slot2:b.bit, slot3:c.bit]]
[phone 2:
  [slot1:d.bit, slot2:e.bit]]
```

When an app is downloaded to a given device, the Cloud RTR system will repackage the application with the set of device-specific bitstreams (possible since the app store has knowledge of a user's device). In Android, for example, apps are packaged in an Android Application Package (APK), which will now include module bitstreams as extra resources for apps that use hardware. To get a rough idea of how this impacts the size of an APK, for the case study we describe in Section 3.7, the hardware module bitstream is 904KB, the Orbot APK of the version we modified is 5.5MB (before any added hardware), and the latest Orbot release is 11MB.

We show that this brute-force approach is quite practical in Section 4.7. As such, it provides a general approach that is deployable and usable today. In addition, we also envision a large amount of reuse of both static designs and hardware modules (*e.g.,* by using precompiled libraries). Just as SoCs are oftentimes reused between different mobile devices, there is no need to have a distinct static design for different devices unless a particular device requires some custom technology.

3.5 Dynamic Module Loading Service

To support apps with hardware, there needs to be system support for loading hardware modules into the FPGA. The operating system will have access to a loading system that can take a hardware module compiled using the Cloud RTR system and load it into the FPGA. However, user applications will not have direct access to this system.

User applications will instead submit requests through a privileged hardware loader system service. Upon loading and initialization of the app, the service will be provided with the location of the app's hardware module files. The service will then choose an empty slot, select the module compiled for this slot, and use the secure loading module to load the module into the FPGA. In the case where no slots are available, the operating system can create 'virtual' slots by time-slicing existing slots. Given the slot reconfiguration time, we do not expect to swap app hardware as frequently as app software, but we see this is as an area for future consideration.

We can implement virtual slots by using the readback capability of FPGAs to store the running configuration of modules, and developers can provide custom unload functionality to aid the readback system in storing difficult to access state (specifically, certain FPGA memory is more difficult to access). Applications that would be disrupted by time slicing can be specifically flagged as unsafe to swap, but the number of these applications running simultaneously should be restricted.

The time to load a hardware module provides an estimate of the time needed to context switch a hardware module. This time is a function of the size of the hardware being written to the FPGA, which we measured to have an average throughput of 37 MiB/s. This leads to a latency of approximately 100 ms for a 4 MB static bitstream, or 27 ms for 1 MB hardware module.

The hardware module is presented as a devfs character device in the Linux /dev directory (when using Android). The hardware loading service will set file permissions to ensure that only the application that requested the loading of the hardware module can access it.

3.6 Evaluation

There are two main questions to answer, which we discuss in this section:

Is there value in user space hardware?

It is generally accepted that hardware will be faster than software[3]. The question we seek to answer here is whether the same performance benefits are retained when we consider it within a system (*e.g.,* does crossing the hw-sw boundary make things worse).

Is the cloud compilation of Cloud RTR practical?

As mentioned, rather than continuing the path of runtime reconfiguration research, which leads to creative, but impractical solutions, we aimed for a solution which was highly practical and deployable immediatly. This resulted in a brute force approach. Here, we ask whether this is itself practical by examining the processing required to support the app market ecosystem.

3.6.1 Application Performance Acceleration

Performance acceleration is one of the benefits of using an FPGA. Of course, we also believe that new applications are now enabled, such as this hardware-based memory scanner. We focus on performance here as a concrete demonstration with a quantitative evaluation.

We focus on three key application domains that are enabled. Each of these applications consists of a hardware module written in C++ using high-level synthesis and an Android application that interfaces with the module. We run the hardware module through the Cloud RTR compilation platform targeting the static design for this development board. The end result is an APK that can be loaded into this demonstration board. Each application fits within these slots, which we defined at 12% of the overall FPGA area – a number resulting from dividing the remaining area after what is needed for the static design by six available slots.

[3]That's not really the focus of this work – we take the stance that there are places where each wins (FPGA, CPU, GPU) and that heterogeneous architectures are good, and more importantly, open programmability is what drives innovation.

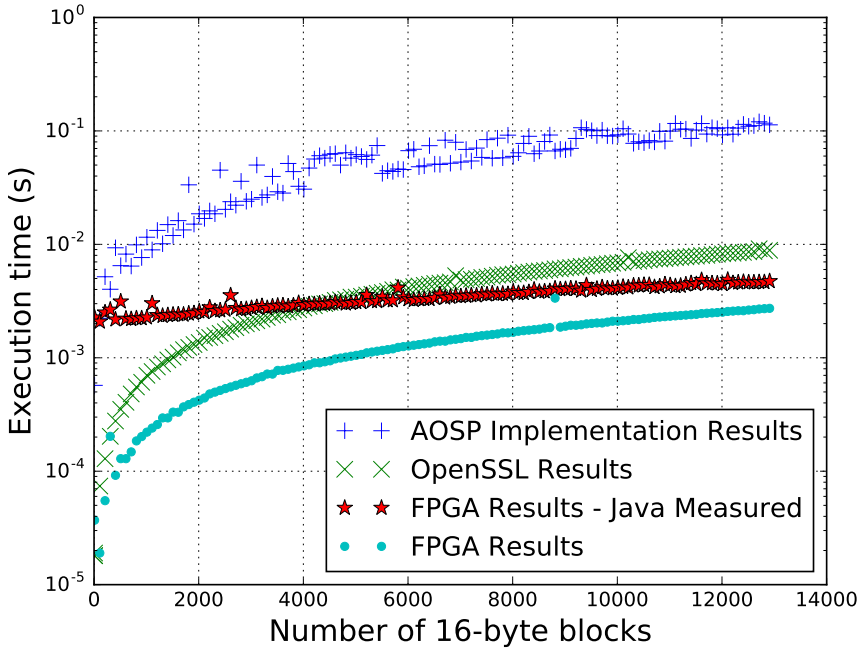

Figure 3.6: The execution time to perform an AES encryption for a range of data sizes – from 10 to 13000 contiguous 128-bit (16-byte) segments of memory.

For these experiments, we prototyped a mobile device using the Zedboard development board, which integrates a Xilinx Zynq 7020 FPGA [11] that has an embedded dual-core ARM Cortex-A9 CPU. We based these Android services on the Android 2.3 and 5.0.2 operating systems that were already ported to our device.

Cryptography: AES

In Figure 3.6, we compare three different AES implementations using our development board, and running Android as the operating system on the CPU[4]:

This figure shows the execution times of the AES implementation (which we derived from [54], though we also experimented with versions from Apple [55] and NIST [56], which had identical performance) for a range of data sizes to be encrypted, varying in size from 10 to 13000 contiguous 128-bit segments of memory. It can be seen that the FPGA implementation is on average three times faster than the OpenSSL implementation, and is approximately 12 times faster than the

[4]We also performed the OpenSSL benchmark in Ubuntu Linux to confirm that the Android OS does not institute a performance penalty.

AOSP. However, the execution time of the FPGA module as measured by Java (marked in red stars) and executed using the JNI is longer than the execution time of the same module when executed directly by a C program (marked in blue diamonds). This is likely due to overhead entailed in copying memory to the JNI function call and transferring control to the JNI. This can potentially be alleviated using Java direct byte buffers passed directly to the JNI function, but is deferred to future work.

Software-defined Radio: QAM

This application can process a signal stored in a contiguous memory region and produce an output signal that is stored into another contiguous region. In a live smart phone, the static design would place the signal off of the antenna into buffers in memory, notify the Android application of a buffer being full, and the application would pass this data to the QAM module. The number of samples the QAM block can process determines the sample rate of the radio application. We implemented this module by modifying (to be compatible with our Cloud RTR system) a reference Xilinx project [57], which comes with C++ code that can be executed in software or run through high-level synthesis to produce hardware.

As shown by Figure 3.7, the hardware implementation is several orders of magnitude faster than the software implementation. The hardware implementation achieves an average throughput of approximately 5 Msps (mega-samples per second), while the software implementation only achieves and average of approximately 500 samples/s. The Xilinx application notes claim a throughput value of 50 Msps [57], which is likely achievable due to the fact that the hardware device is intended to process data received directly from an analog-to-digital converter (ADC), whereas our implementation has been retrofitted to stream data from system memory.

Memory Scanner

Our final application is a simple implementation of a hardware memory scanner that searches our device's address space for occurrences of a 16 byte strings.

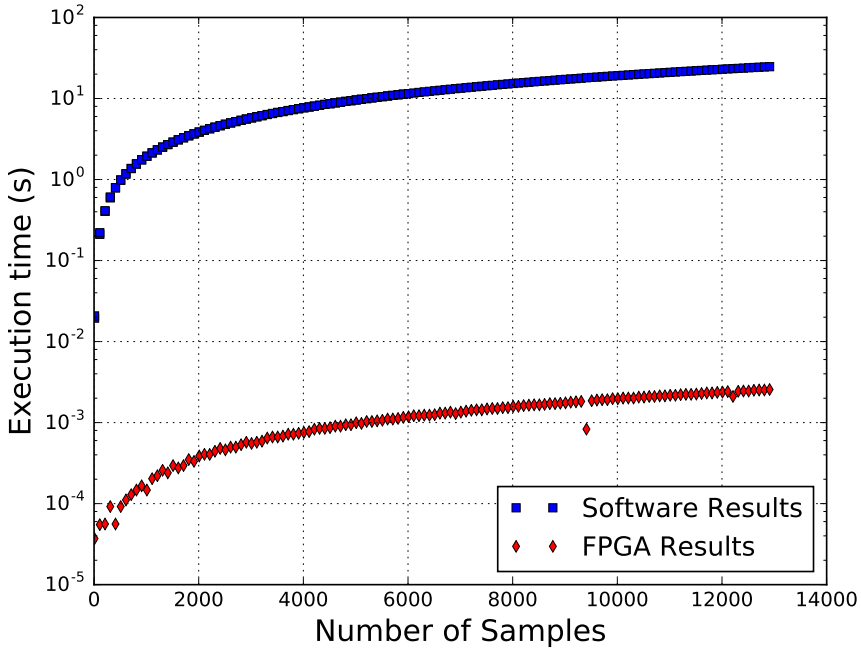

Figure 3.7: The execution time to process a different number of samples with my QAM application.

Using the LMbench testbench [58], we instituted a memory benchmark that measured the throughput of our device's memory while under normal operation and while the hardware memory scanner was executing (which is constantly reading from memory). Using this benchmark, we measured a 2.7% reduction in performance for read operations and a 5.5% reduction in performance for write operations.

3.6.2 Cloud Compilation Resources Needed

We propose performing compilation of the reconfigurable module in the cloud as part of the process to upload to the app store. To understand the feasibility of this, here we evaluate the amount of computing resources needed to sustain a 'user space hardware' ecosystem.

The metric of interest is how long it takes to compile a single reconfigurable module for a given static design. Recall that a static design is the base design that roughly corresponds to a system on chip used for a given smart phone. These static designs have open areas (slots) for placing reconfigurable modules.

For all experiments, we used a server with a 6-core Intel Xenon CPU (2.1 GHz, 48 GB RAM).

# of slots	Compilation Time (min)	Throughput (apps/day)
2	11.92	121
3	14.93	96
4	19.02	76
5	24.21	59
6	28.23	51

Table 3.1: Compilation time and number of apps a single server could service per day.

Single App for a Single Static Design

An app with hardware uploaded to the app store must have its hardware modules compiled for each slot that it can be placed into. However, certain steps in the process do not need to be performed for each slot – *e.g.,* synthesis needs to be performed once for each module, then for each slot, the synthesized module needs to be placed and routed. For this evaluation, we used an FFT module, which is a highly regular structure and enabled us to adjust its parameters to effectively alter its size to fill up any slot size we experimented with.

Table 3.1 shows the total time to compile a single application's reconfigurable module for static designs with two to six slots (each slot is defined as 12% of the overall area of the FPGA, as previously mentioned), as well as the extrapolated throughput (the number of apps that could be compiled per day on one server given this compilation time). We chose up to six slots (in contrast to the 60-100 slots in [28]) as we believe each to be big enough to implement a reasonable module within a single slot.

Compiling All Apps

Using the calculated throughput, we can now estimate the amount of computing resources needed to service the entire app ecosystem.

First, we need to know how many apps are uploaded each month. The company AppFigures provided us with the Google Play Store application upload figures for the entire year of 2017, with a total of 1.5 million apps at the end of 2017. Based on this, we calculate the approximate monthly

6 Slots Requirement	% of Jan 2017 Apps that Use Hardware		
	0.1	1	10
	# of Apps Uploaded per Day		
	5	50	500
# of Static Designs	# of Machines Required to Compile RMs		
1	1	1	10
10	1	10	99
100	10	99	981
1000	99	981	9804

Table 3.2: Number of servers required to support the compilation requirements of Cloud RTR, assuming designs with six slots.

upload rate for the beginning of 2018 to be 125,000 apps per month.

Table 3.2 shows the number of machines required to service monthly demands for compiling apps with hardware, for six slots as an example (2-5 slots would be proportionally less). Each table varies the number of apps with hardware uploaded each day based on the percentage of applications that require hardware (0.1%, 1%, and 10%), as well as the number of static hardware variants – for the sake of illustration, we assume from one to 1000 variants, with each interval increased by a factor of ten (we expect the number of variants to be on the low end, as static bitstreams can be reused between devices, just as phones today use a small set of SoCs, and not every device will require a new static bitstream).

The cloud provider will easily be able to support the lower end of the spectrum internally. On the upper end, the cloud provider might look to relieve the computation burden by offloading to the phone manufacturers to compile for their own variants.

3.7 Case Study: Orbot Tor Client

Developing our demonstration applications from scratch allows us to design it to use an FPGA natively. Here we explore modifying an existing, complex application to make use of a hardware module to understand the inefficiencies that may result.

We chose to modify the Orbot [31] Tor [32] client for Android. Tor is an anonymization network that allows a user to access the internet without disclosing their source IP address, making identifying and tracking their internet traffic very difficult for third parties. A Tor client creates a circuit through this network and encrypts their traffic separately for each node along the path to prevent eavesdropping during transmission. Because of this extensive use of encryption and based on notes by the Orbot developers mentioning that AES is one of the areas to optimize Orbot [59], we see this as an ideal case study.

Our AES module implements the CTR (counter) mode of operation on top of a standard AES block cipher that is an equivalent to the OpenSSL CTR implementation used by Tor. In order to integrate this AES accelerator with Tor, we replaced all calls to OpenSSL AES encryption with calls to the FPGA accelerator, which proved to be a fairly minor modification. We also needed to ensure that all data that was to be encrypted was located in a contiguous memory region with a known physical address, which required us to replace all *malloc()* calls with calls to a custom memory allocator and leverage a memory region that we reserved from the kernel.

Thus modified, the application is able to make use of the FPGA resources and operate correctly. However, there are inefficiencies remaining due to (i) the overhead required to allocate memory in the reserved region, (ii) the overhead in accessing this memory, as it is implemented using memory-mapped I/O, and (iii) the fact that certain memory system calls (*e.g., malloc()*, *memcpy()*, and *memset()*) are incompatible with the current memory mapped implementation – which would require more extensive modifications to the code to resolve. Even so, this provides us with great insight into how apps should be designed to capitalize on the FPGA resources and is an area for future improvements.

3.8 Related Work

Although there are no existing systems that implement all of the functionality of our Cloud RTR system in mobile devices, work has been done in reconfigurable computing in other contexts, including several different attempts with Android.

Of note from previous reconfigurable computing research is the BORPH system [49], which attempts to create operating system extensions in Linux for FPGA operations, and uses Berkeley's BEE2 system [60], and the more recent Connectal framework [61], which can automatically generate hw-sw interfaces during hardware development. These systems do not, however, address application distribution or FPGA resource sharing.

In terms of mobile systems research, some proposals have been made, such as the rSmart system [62] and the work from Smit et. al. [26]. Smit et. al. proposes a similar hardware architecture to the Zynq-7000 architecture, but does not present an operating system integration or a deployment system. The rSmart system only presents a high-level sketch of a system similar to ours, but no details on implementation or integration are provided. Our system builds upon this research to create a general system that is deployable using existing technology.

There has also been recent advances in reconfigurable cloud platforms. For example, Microsoft's Project Catapult makes use of FPGA peripherals in data centers to accelerate web searches [63] and neural networks [64], and Intel's acquisition of Altera [65] is leading to x86 CPU architectures coupled with FPGAs [66]. Microsoft's solutions, however, are only single-application hardware accelerators, whereas our system allows for usage in general applications. Intel's system is more general, but has not been released publicly, although it does claim to use OpenCL [67] as the software interface.

Our work is complementary. For example, OpenCL can be used on mobile devices with support from major hardware manufacturers, such as ARM, Intel and Qualcomm, [68, 69, 70, 71, 72], and can even be used with our system by using a compatible hardware module. OpenCL's main limitation is its focus on parallel acceleration, which does not enable new architectural enhancements, such as our SDR or security applications.

Reconfigurable Android devices and systems have also been proposed, such as Google's Project Ara [25], among others, including various other modular phones [73, 74, 75]. These modular phone systems allow for reconfiguration and upgrading of smart phone *physical* components, similar to how personal computer components can be upgraded. However, these modular architectures can only be reconfigured manually by the user replacing the physical modules, whereas our system allows for dynamic and custom reconfiguration by software.

Finally, the Android OS has been ported to the Zynq-7000 in several projects, such as the work of Barbareschi, et. al., among others [27, 76, 77, 78]. However, with the exception of the work of Barbareschi, et. al., these projects only port the OS to a new device. The work of Barbareschi, et. al. only extends this work to create an Android-compatible custom accelerator to address a single problem, whereas our system allows for any general software developer to create their own custom hardware modules.

Chapter 4

SDSHW: Enabling (Programmable) Secure Hardware

In Chapter 3 we described how we overcome the challenges in sharing an FPGA to enable user space hardware. In this chapter we describe how we overcome the challenge of programmability to enable secure hardware on an FPGA. It is silicon's property of immutability that is traditionally relied upon by secure hardware, but the programmability of FPGAs seems to negate this property. Just as Cloud RTR provides a means for application developers to provide their own implementations of hardware for their applications, here we enable developers to provide their own implementations of *secure* hardware. We overcome the problems presented by the fact that FPGAs can be reprogrammed to allow for this hardware to be used in an FPGA while maintaining the same security as if they were implemented in traditional silicon.

4.1 Introduction

Secure hardware provides many benefits for securing systems due to the immutable nature of silicon circuitry, whose functionality cannot be altered after it is manufactured. As previously highlighted, secure hardware systems can implement various applications, *e.g.,* brute-force resistant user data encryption [1], authenticated data feed systems [2], scalable blockchain transactions [3],

Feature	TPM	TZ	SGX
Flexible Root of Trust	●	●	○
Trusted Execution Environment	○	●	●
Remote Attestation	●	○	●
Peripheral Access	○	●	○
Trusted Input	○	◐	○
Hardware RNG	●	○	●
Hardware Crypto	●	◐	◐
Secure Storage	●	○	●
Shared Architecture	◐	●	●
Oblivious Memory	○	○	●
Cache Side Channel Defense	●	○	○
TLB Side Channel Defense	○	●	○

Table 4.1: **Secure Hardware Features** — We compare the features supported by Trusted Platform Modules (TPMs), ARM TrustZone (TZ), and Intel SGX. ● represents support, ◐ represents partial support or support that depends on how the design is instantiated, and ○ represents no support.

and protected cloud computing [4], that derive their security from the properties of silicon hardware.

Despite the benefits that these types of applications can provide, we are currently stuck with a constrained ecosystem of secure hardware providers. Due to the constraints that silicon manufacturers and hardware assemblers must operate under (which are essentially the same as the constraints described in the previous chapter) – cost, time, and complexity of hardware design and manufacture [79, 80] – the design choices and trade-offs are decided unilaterally by a small number of these manufacturers and assemblers. This results in scattered support of a wide range of features, and ultimately limited selection for users of secure hardware. Table 4.1 presents a summary of several secure hardware systems and the features they support. Even in this modest list of features, there is no existing system that offers every feature, despite each system implementing features the other does not.

For example, while Intel Software Guard Extensions (SGX) and ARM TrustZone both provide Trusted Execution Environments (TEEs) that allow applications to execute trusted code in an isolated component, only Intel supports Remote Attestation of software running in the TEE. However, Intel made the decision that they must be the trusted party to provide proof and verification of the

remote system's state. Furthermore, updates to secure hardware systems in response to discovered vulnerabilities [81, 82, 83, 84, 85, 14, 15, 16] or demand for new features are at worst impossible, and at best gated by the chip manufacturers, leaving system designers that use secure hardware at the mercy of a few companies.

These design trade-offs and features may make sense for some applications, but the ultimate problem is that these decisions are made by the silicon manufacturers, and not by the individual developers that ultimately use secure hardware in their design. In order to realize our ultimate vision of user space secure hardware, we need to change this design flow so that developers have control of the secure hardware features that are available. This enables secure hardware to be customized to different applications.

In this chapter, we propose empowering the individuals that ultimately use secure hardware to make the decisions that are right for their needs. We introduce an architecture we call **Software Defined Secure Hardware** (SDSHW), that enables system provisioners and application developers that wish to use secure hardware functionality to develop custom logic, deciding for themselves what sets of features they ultimately need in their design *without sacrificing the security properties of fixed hardware*. This provides us an essential building block for our user space secure hardware vision, namely a secure platform for custom, application-specific hardware. In the previous chapter we provided a solution for making hardware into a user space resource for applications. Here we provide a similar capability that lets application developers implement secure hardware for their applications.

As we have shown in the previous chapter, FPGAs can be used as a resource for software applications, which enables developers, rather than requiring manufacturers and assemblers, to decide what hardware features their applications need. Here, we want to enable a similar model for secure hardware developers by leveraging the reprogrammability of FPGAs.

While prior researchers have proposed building open source secure hardware features such as remote attestation and trusted execution environments [86], and others have leveraged FPGAs for cryptographic primitives [87], they only demonstrated that it is possible to implement security

functionality on FPGAs. Others have implemented more extensive security functions in FPGAs and have even addressed some of the threat posed by advanced physical adversaries (*e.g.,* DPA resistance) [88]. However, none of these solutions provide a means to allow for *arbitrary* secure hardware; even the solutions using an FPGA cannot provide all the properties of a secure hardware system while still allowing reprogrammability of this hardware. SDSHW enables the ability to design arbitrary *secure hardware* with a flexible root of trust on top of the flexible platform by breaking the traditional trust relationships that were described in Chapter 2.

With all this previous work, however, we ask the question: Why hasn't this been solved before? Previous research has implemented various different secure hardware systems and has used FPGAs to provide security functionality, so extending them to provide the complete security properties would appear to be trivial. However, we identify that prior research leverages processes we remove, as our system breaks these process trust relationships that exist in traditional secure hardware. These processes enable prior work with FPGAs to achieve the needed security to provide their applications, but their presence violates the properties of secure hardware when run on a reconfigurable platform.

To summarize this process trust: prior work in use of FPGAs for secure hardware requires trust in a provisioner or developer whose hardware design is executing in the FPGA, and this party can change the configuration at will. The hardware in the FPGA therefore has to trust the processes that the provisioner maintains for both the hardware to be secure and for it to even provide its function. This differs from traditional secure hardware process trust, as the functionality of the hardware still cannot be changed, even if the process trust is violated. With an FPGA, the process trust and the functionality trust are linked, as the functionality of the FPGA can be changed after it has been programmed at any time.

We remove this process trust in SDSHW by putting the device, rather than an external party, in control of its own reconfiguration. This prevents hardware from being overwritten or modified without detection, meaning that it can maintain the properties of functionality and isolation, since updates will always be detected and will require approval from the device to be applied. We

provide this with SDSHW's novel self-provisioning system, which leverages a minimal amount of fixed silicon hardware commonly found in FPGAs to ensure that only authorized hardware can exist in the FPGA. On top of that, we provide an update protocol that requires authorization from the FPGA before new hardware can be authorized based on security policies established during provisioning, allowing for the reprogrammability of the FPGA to still be used.

The removal of process trust enables SDSHW, as we can now use the reprogrammability of FPGAs to provide secure hardware applications with the same properties as hardware. This enables system provisioners and application developers to take control over the design decisions of secure hardware implementations by giving them the freedom to chose the features required to support their applications, as any device can be used to support any feature. As we can implement SDSHW on commercial-off-the-shelf (COTS) hardware, developers are further empowered to design their own devices if existing system provisioners refuse to provide a feature in an existing device.

In the rest of this chapter, we will describe the design of the SDSHW platform and the implementation of our proof-of-concept applications, along with an evaluation of their performance. Specifically, we make these contributions:

Provide secure hardware self provisioning: In order to provide the security properties of hardware in an FPGA, the designed functionality of a secure hardware application must be fixed to only provide this application without any vectors for observation or interference. To provide this, we have implemented a method for FPGAs to provision themselves so that only authorized hardware can exist in the FPGA and that the FPGA is isolated from the rest of this system, thus providing these properties.

Identify Minimal Fixed Hardware: In order for the self-provisioning system to be able to configure the FPGA to provide the needed properties, the actions of the self-provisioning system need to be permanent and unable to be circumvented. To do this, we have identified a minimal amount of fixed secure hardware that enforces the configuration of the self-provisioning process. This effectively allows the self-provisioner to extend the properties of this fixed secure hardware

to the reprogrammable hardware; if this fixed hardware remains secure, so will the FPGA.

Create the SHSHW platform: We have designed a platform that allows secure hardware to be implemented in an FPGA. The Software Defined Secure Hardware platform leaves the choice of features to implement up to the system provisioner and application developers, allowing for flexible roots of trust, easy extension, and security updates.

Implement SDSHW on COTS hardware: To demonstrate the practicality of SDSHW, we have developed a prototype implementation on a COTS SoC. We then prove the flexibility our framework by implementing several secure hardware modules and applications and evaluate these applications to determine the necessary performance trade-offs that must be considered when implementing secure hardware on our platform.

4.2 Related Work

SDSHW allows application developers and system provisioners to have control over the secure hardware features that are available in a device. As mentioned previously, there has been significant prior research on secure hardware in various forms, from different secure hardware features to improvements on existing models, and even work that has leveraged FPGAs to varying degrees. Here, we differentiate ourselves from this prior research and explain how none of this technology can provide the same capabilities as SDSHW.

4.2.1 Software-based Solutions

Due to the difficulty of designing and implementing secure hardware in silicon, many systems rely on software to define much of the functionality of a system, such as firmware and CPU microcode [89, 90]. However, other systems go further by designing special isolated environments and even exposing these systems to run arbitrary software. However, these systems are essentially

software built on top of a secure hardware foundation. Though the software can be updated to modify the system's security, the underlying platform cannot.

4.2.2 Secure Coprocessors

A secure coprocessor is a separate execution environment for software that is isolated from the rest of the system by the physical design of its interconnects. These systems execute some software that can securely generate and execute upon secret state, as the hardware prevents this state from being observed. The most common example of these systems is the Trusted Platform Module, which is a secure coprocessor that executes a software application that presents an industry-standardized API [91, 92]. These APIs allow systems to request cryptographic operations to be performed using keys stored securely in the TPM, allowing tasks such as encryption to be performed without risk of exposing these keys. Such capabilities are essential to data protection as is used by disk encryption in Chrome OS [93], Microsoft Bitlocker [94], and Nokia mobile platforms [95].

Although few problems have been found in the specification of TPMs, the rigor in designing the specification has caused problems for its usefulness, and has even caused applications to seek alternatives [96]. The time required to implement new TPM hardware meant that an implementation was not available when needed by an application, which forced Microsoft to re-implement the TPM spec in a trusted execution environment. This shows that secure coprocessors can provide the needed security with a software-based approach, but that the inflexibility of the system's feature sets is still a problem. SDSHW would have solved this problem by allowing for an existing hardware device to be updated to support a new standard, rather than requiring new hardware to be manufactured.

4.2.3 Trusted Execution Environments

A more flexible approach than existing secure coprocessors is to isolate arbitrary code from the rest of the system. The two most prominent implementations of TEEs are ARM TrustZone and

Intel SGX, both of which implement an isolated execution environment on an otherwise standard CPU [97, 80]. Though the method for loading code into these systems differs, each allows the isolated code to be interacted with using special CPU instructions, allowing the code to implement its own API and present it as a service to other software in the system. For example, Microsoft has used TrustZone to implement a firmware-based TPM when updated implementations of the TPM specification were unavailable [96].

These systems are not without problems, however. ARM TrustZone lacks both a method to store secrets and a means to ensure that the correct code is loaded into the secure environment. These features must be provided by separate secure hardware systems, such as secure boot (to authorize correct code) and replay-resistant storage (to securely protect persistent state) [96]. However, neither of these features is provided directly by TrustZone and they are not commonly found together at the same time in ARM systems that are TrustZone enabled. Microsoft was able to solve this by designing their own hardware and partnering with manufacturers, but other application developers do not have this option.

SGX has been well studied to reveal potential attacks on its implementation due to its more complete feature set and its availability on traditional desktop and server CPUs, which makes it both more interesting and easier to study. Research has found many vulnerabilities in SGX's implementation and have proposed software patches on SGX programs to work around them, such as to address problems from cache side channels, I/O side channels, and control channel attacks that use page faults to affect execution [98, 99, 100, 101, 102, 103, 104, 105, 106]. Work has also been done to allow for new features to be added to SGX or to overcome hardware limitations: Komodo decouples hardware mechanisms from management using a privileged software monitor [107] and SGXIO attempts to bring trusted I/O to SGX [108].

All these systems build on top of a complex hardware system (the TEE hardware) to provide their security functions, but inherit the vulnerabilities of these platforms when they are discovered, or have difficulty with implementation due to missing features. Developers are in general powerless to change any functionality of the platforms beyond redesigning their application software, which

can only go so far and cannot provide a missing hardware feature. SDSHW can be used to allow developers to make these needed changes rather than being reliant upon silicon manufacturers to fix vulnerabilities and provide new features.

4.2.4 Hardware-based Re-designs

Another research direction simply proposes new processor designs. Sanctum, a RISC ISA extension, mitigates software side-channels and protects DRAM access [86]. Aegis provides secure DRAM usage to the TEE using encryption and integrity verification [109]. XOM maintains sensitive code and data in isolated containers, however, it fails to protect from memory access pattern attacks [110]. Bastion provides a hardware supported secure hypervisor environment, protecting hypervisors from guest VMs using virtual memory compartmentalization and secure storage [111]. Hyperwall extended this idea to provide full protection for guest VMs from potentially malicious hypervisors by preventing direct memory access once a VM is provisioned and providing signed evidence of memory confidentiality [112]. Iso-X offers the same properties as SGX, and protects DRAM access using a section provisioned at boot, however, the proposed ISA changes add to the core's cycle time and do not protect against cache timing attacks [113].

These solutions are only addressing point problems with secure hardware, adding missing features for specific applications without addressing the overall problem of features sets (just adding another column to Table 4.1 with a different set of filled in and empty circles). SDSHW tackles this problem directly by allowing application developers to select the needed features, meaning that any combination of these solutions could be put together in a system, rather than requiring new silicon to be manufactured for each application.

4.2.5 Programmable Co-processors and FPGA Solutions

The avenues of prior research most closely related to SDSHW are the work on programmable co-processors. For example, the SAFES architecture demonstrates the use of FPGA components to provide security primitives and guarantee invariants in program execution [87]. These works break

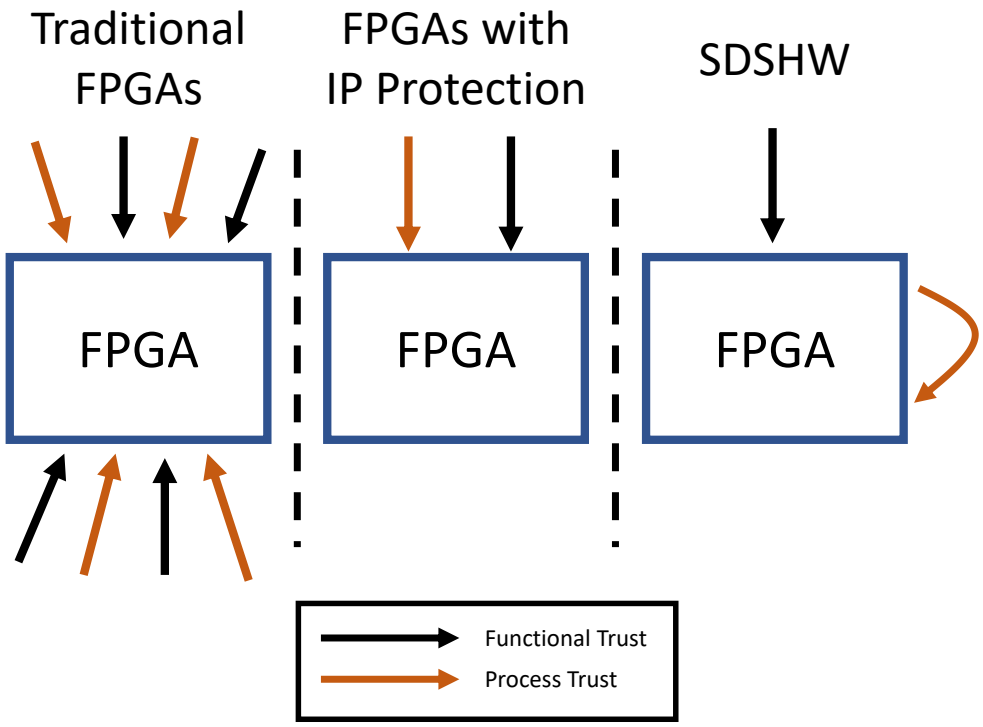

Figure 4.1: **Process Trust** Process trust and functionality trust are the two different trust relationships we identify. Functional trust is a one-time promise to implement the functionality of a system, whereas process trust is the continued trust of an entity for the system to operate correctly. Traditional FPGAs can be reprogrammed from any source, and so functional and process trust are not differentiated. When used with IP protection and anti-tamper technology, FPGAs lock their process and functionality trust to a single source. In SDSHW, self-provisioning allows for a single source of the initial functionality, and the update system allows for the device itself to mediate process trust.

down into primitives implemented in FPGAs [114, 115, 116], or programmable designs [117, 118, 87]. These systems make use of an FPGA to allow for hardware solutions to be developed more quickly without the need to manufacture new hardware, and even take steps to isolate this hardware. However, they do not actually provide the same properties of physical hardware, which SDSHW does by preventing arbitrary reprogramming of the FPGA.

More advanced uses of FPGAs, such as products offered by Microsemi, offer a more secure platform for running secure functions by using advanced fixed hardware blocks to implement sophisticated intellectual property (IP) protection [88]. Their products are also resistant to DPA attacks, and are locked to allow only a trusted developer's applications to execute in the FPGA.

However, these systems do not go far enough to provide the same security properties as hardware, as a hardware application cannot have any expectation of fixed functionality or isolation due to the inherent process trust relationship with the trusted developer. SDSHW removes these process relationship so that FPGA hardware can provide these properties.

The differences between these uses of FPGAs that provide secure hardware are shown in Figure 4.1. Security primitives implemented in FPGAs have many process trust relationships, as they can be updated at any time (the leftmost example). The use of IP protection technology (*e.g.,* the Microsemi approach, in the center) reduces these process trust down to a single relationship with the developer. SDSHW uses both IP protection and self-provisioning to allow the device itself to mediate the process trust (the rightmost example); the application may choose to still export this to an external entity, but it is not inherently required. It is this change of process trust that allows for SDSHW to provide the same security benefits as hardware, where prior solutions cannot.

4.3 Architecture

As illustrated in Figure 4.2, our SDSHW platform only requires a small amount of fixed hardware, which we leverage to provide a platform that allows for custom secure hardware to be built in programmable logic, with the same properties as secure hardware. In this section, we describe the hardware requirements and components of our architecture.

4.3.1 Fixed Hardware Requirements

We have already identified fixed functionality and isolation as the two properties of hardware that we are trying to provide with SDSHW. As was described in Chapter 2, one of the overall goals of user space secure hardware is to maintain the same threat model of secure hardware while still providing these properties.

SDSHW provides these properties while still using the reprogrammability of an FPGA, but these same properties are seemingly incompatible with reprogrammable logic: FPGAs do not

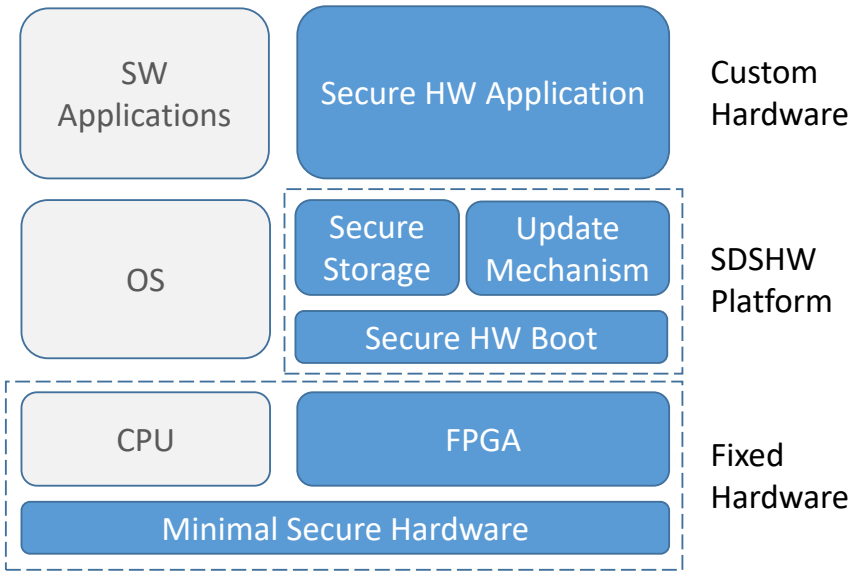

Figure 4.2: **SDSHW Stack.** The SDSHW platform is built on top of the CPU/FPGA SoC, where the trusted (blue) secure boot is responsible for loading the SDSHW logic into the FPGA at boot, and protecting it from the untrusted (grey) CPU and software stack. The SDSHW platform provides several secure hardware components useful for modules such as trusted execution environments, and remote attestation, that are built as applications on top of the platform.

provide fixed functionality, as they are designed to be reprogrammed, and their isolation depends on the interface with the rest of the system. We show how to provide these properties in an FPGA by leveraging a minimal amount of fixed secure hardware present in existing SoCs to ensure only the trusted secure hardware is present in the FPGA, and that it is configured to be isolated from the rest of the SoC.

Our platform achieves security by bootstrapping the FPGA into a secure state and authorizing an initial hardware design to execute in the FPGA. A self-provisioning process configures the secure hardware to isolate the FPGA from the rest of the system and prevent unauthorized hardware from being loaded into the FPGA. The initial configuration that is authorized by the self-provisioning system may include an update subsystem that can authorize future updates based on a security policy established at provision time, which is also encoded in the application's logic. Future updates are therefore possible, but only if they conform to this security policy that is enforced by the hardware in the FPGA. The fixed secure hardware prevents any other hardware from ex-

isting in the FPGA and prevents all external interference with this hardware, thus providing fixed functionality and isolation, and also ensures that the security policy implemented by the update system cannot be circumvented.

Essential to the operation of SDSHW are specific requirements of this fixed secure hardware in any device, as these systems are used by the self-provisioning system to protect the FPGA. Specifically, we require the following fixed hardware capabilities:

1. Anti-tamper logic that can isolate the FPGA from external access and reprogramming that can be disabled after the FPGA is configured at system startup.

2. A secure boot technology that is capable of configuring the FPGA with an authorized hardware configuration, based on verifying a signature over potential configuration bitstreams at boot.

3. A secure storage system accessible only to the FPGA that can be used to store the authorization keys that are generated during self-provisioning and programmed into the secure boot system. The keys are needed in future for an update subsystem to authorize updates.

These requirements can be found in most COTS FPGA systems, as SDSHW reuses the secure hardware systems used for IP protection. SDSHW uses them in a different manner, however, as we break the process trust that these hardware systems are designed to enforce.

4.3.2 SDSHW Platform

In the previous section, we described the minimal fixed hardware needed to enable our architecture and how it is used to provide the security requirements of our platform. In this section, we describe the three main features that allow for SDSHW to operate and how they interact to provide the complete platform. These features are the self-provisioning system, the update system, and the secure storage system, that when combined, comprise our platform. We describe an implementation of applications built on this platform using a commercially available device in Section 4.4.

Self Provisioning

The self-provisioning system is essential to providing the security properties of secure hardware, as the FPGA needs to be configured into a secure state to prevent running applications from being compromised. Once this state has been established, any hardware configuration running in the FPGA will have these properties: that of fixed functionality and isolation.

The secure boot system is used to provide and enforce these properties. We note that traditional secure boot systems can be circumvented if the private key used to authorize the booted software is leaked [14, 15, 16]. This results from the inherent process trust that is used with most secure boot systems: the keys used to authorize new booted data are held by a party external to the system. The secure boot system is therefore dependent on this party for its security to be maintained; even though there are no implementation flaws in the system, this trust can be abused, as has been shown. However, we have stated that one of the contributions of SDSHW is that we break these process trust relationships.

The self-provisioning system does this by generating all secure boot keys inside of the device, rather than leaving them in the hands of an external party. This breaks the process trust relationships of secure hardware that rely on the silicon manufacturer to maintain state as these relationships are defined as the external party having control of this key, which is shown in Figure 2.1. As prior incidents have shown, these relationships can cause the security of secure boot to be removed without any flaws existing in the hardware itself. If this were to occur in SDSHW, then an unauthorized FPGA configuration could be loaded by compromising secure boot using these relations, thus also negating the property of fixed functionality that we provide. Therefore, SDSHW has to break these relationships to meet our stated goals, which is achieved by generating the keys internally.

Secure boot systems typically operate by verifying a signature over code at boot. The device maintains the public keys, while an external party keeps the private keys to allow them (and only them) to sign updates. In SDSHW, the self-provisioning system generates these keys on the device and programs the public key to the secure boot system and signs an initial hardware configuration.

The system provisioner is responsible for providing this self-provisioning system and selecting this initial hardware configuration, which is illustrated as the additional functional trust in Figure 2.1.

It is assumed that the secure boot system will only load a correctly signed hardware configuration into the FPGA, meaning that only a configuration signed by this key can ever exist in the FPGA. Therefore, to support the update system of SDSHW, the generated key must be retained, and so is stored in the secure storage system. Assuming that the secure boot system cannot be circumvented or disabled, as we require, the only way to load an unauthorized configuration directly into the FPGA would be to break the cryptography used by secure boot, such as by generating an RSA signature without the correct private key.

The self-provisioning system must also ensure that the FPGA is isolated from the rest of the system to provide the fixed isolation property. This is simpler to achieve in theory, as most FPGAs contain anti-tamper logic to protect running bitstreams. The self-provisioning system must either set system configurations that prevent reconfiguration and readback access to the FPGA after boot, or ensure that these settings are set each time the system boots. This can be done by configuring global system registers or by signing a trusted bootloader (or other boot-time code) that will always perform these actions. Unfortunately, configuring these systems is very device-specific compared to secure boot systems, and so a specific sequence of actions cannot be prescribed for all situations. The system provisioner is responsible for selecting or customizing the self-provisioning system so that it is targeted to a particular device, and is therefore designed to set up the system's anti-tamper logic correctly for how it is implemented in that system.

Secure Update System

The self-provisioning system essentially 'locks' the FPGA to a single hardware configuration at provisioning, meaning that no other secure hardware implementations can exist in the FPGA after is has been provisioned. However, one of the stated goals of SDSHW is to allow for updates to be provided if they conform to the application's security policy. This security policy is determined by the application designer and encoded as a subsystem in their application logic that is locked

to the FPGA during provisioning. As no other logic can exist in the FPGA after this point, this effectively means that this security policy is established at provisioning and is enforced by the secure boot system, which prevents any other policy from existing.

The mechanics of performing an updates are simple. An update is authorized by reading the secure boot key from the secure storage system and signing the update, which is simply a different FPGA bitstream from the one currently executing. This signed bitstream then replaces the current one in the persistent storage media that the secure boot reads the boot data from. After the signature has been generated and written, this new hardware can be booted on the next device power cycle.

The application developer must therefore implement this functionality in their update subsystem. However, they must also select the conditions for when an update will be accepted, in the form of a security policy. Examples of these policies are:

1. Signed by the developer's key.
2. Authorized by the end user.
3. Signed by the developer and authorized by the user.
4. No updates ever allowed.

For example, if the third option is chosen, the developer must implement the update subsystem to verify a signature based on a key encoded into the application logic and receive some sort of input from the user. This can be as simple as the pressing of a physical button to more complex authentication, such as the input of a PIN or password that is compared to a value stored in secure storage.

The application developer must make the choice about which security policy to encode into the logic, however, if updates are to be allowed. SDSHW does not provide an implementation of this system, as no one security policy will meet the needs of all applications. As the secure boot key is stored in the secure storage accessible only to the FPGA, if no update mechanism is implemented to perform these checks and then read the key, no updates can be performed.

This update system effectively puts the FPGA in control of updates and introduces a new process trust model, where the security of the FPGA depends on this security policy that the FPGA itself enforces, meaning that the FPGA must trust itself to maintain its own security. This security policy can outsource this trust to a third party by requiring an external signature, but the device still mediates this trust. This allows updates to occur without compromising the property of fixed functionality; no update can be made to the device without its involvement, meaning that any application in the FPGA will be notified if its functionality is changed. Therefore, any application in the FPGA can always be assured of its own functionality when it is running, as no update can occur without its knowledge. When updated, this property still exists, but now a new functionality is covered by this property.

The update system takes advantage of this self-provisioning to allow a new process trust model, in which the device mediates its own updates. The only way for a new hardware configuration to be booted on the device is for it to be signed with the same key as is programmed into the secure boot hardware. This key can only be accessed by the FPGA after provisioning, meaning that only the FPGA can authorize a new update. Any hardware configuration in the FPGA will need to include an update subsystem in its logic that will sign a new update if it conforms to the specific security policy that is needed by the application. As this security policy is encoded into the actual implementation of the update system, and this update system is part of the initial FPGA hardware configuration, it is effectively established at provision time and cannot be overwritten or circumvented, so long as the secure boot system cannot be compromised.

Even though SDSHW does not prescribe a specific security policy, the steps that are required to perform an update are identical, other than the enforcement of the security policy. However, the exact methods for passing data to an FPGA and accessing the secure storage will vary for different devices, and so it is impossible to provide a single implementation for all of these cases (the FPGA design process only allows implementations to be targeted to a single device in any case). We do provide an example implementation of an update system in the next section, showing that providing such an implementation is practical and does not require significant FPGA resources.

Secure Storage

The secure storage system is the final component of the SDSHW, and is essential for the update system to operate. However, its design is not complicated and is also dependent on the capabilities of the target device. We require some method to securely store the secure boot keys that are generated during self-provisioning, as these keys are needed by the secure boot system. It is assumed that a secure storage system exists that is only accessible to the FPGA, but the actual interface to this system is dependent on how it is implemented in that device. For example, the FPGA may have access to a dedicated flash storage device, or may have to share access to a persistent storage media with a coupled CPU. All that is required is for the secure boot to be stored in a manner that only the FPGA can access it, which can be achieved by encrypting it and storing it in persistent storage. An encryption key can be encoded into the FPGA in a number of ways, including using bitstream encryption, partial reconfiguration, and physically unclonable functions (PuFs).

The self-provisioning system is responsible for storing the secure boot keys after they are generated, meaning that it must also initialize this storage if necessary. If only a shared persistent storage device is available, then the self-provisioning system is responsible for encrypting the secure boot keys in a manner that allows the FPGA to access them. This means that any secure hardware application for a device must be aware of the device's capabilities so that application designers know whether an encryption key needs to be generated on the device and encoded in some way. The self-provisioning system will be aware of these requirements and will perform the necessary encryption, possibly by loading the initial hardware into the FPGA to allows for it to use its encoded key to perform the encryption at this stage. This implies either a standardized method of implementing secure storage, or close interaction between system provisioners and application developers. As application developers must already trust the provisioner and interact with them in order for their application to be included as the initial hardware in the FPGA, we do not see this interaction as being a barrier to usage.

4.3.3 SDSHW Threat Model

As presented in Chapter 2, the goal of SDSHW in terms of threats is to maintain the same threat model as secure hardware implemented in silicon, *i.e.,* to defend against the same adversaries and provide the same protections. That chapter defined the properties of secure hardware as fixed functionality and isolation, which our self-provisioning system attempts to provide.

The self-provisioning system makes several assumptions:

1. The FPGA secure boot system will not let an unauthorized (*e.g.,* unsigned) bitstream be loaded in the FPGA.

2. The FPGA anti-tamper logic can isolate the FPGA and cannot be circumvented.

3. The secure storage protects the stored key from being read outside of the FPGA.

4. The FPGA silicon itself is implemented to not have backdoors that can compromise the running bitstream.

The assumptions essentially require trust in the silicon manufacturer to have implemented the FPGA correctly and non-maliciously. The secure boot hardware enforces that only the initial hardware authorized during self-provisioning can exist in the FPGA, meaning that this bitstream has the property of fixed functionality – only this hardware can exist in the FPGA and cannot be modified. This assumption holds so long as the secure boot system is implemented correctly (*i.e.,* there is no other way to boot a bitstream in the device once the secure boot system is enabled) and the cryptography it relies on is correct. For example, if the secure boot verifies an RSA-4096 signature of a SHA3 hash of the bitstream, the system is secure so long as this cryptography cannot be broken. As the secure boot key never leaves the device, there is no other way to generate a valid signature than from within the FPGA, meaning that a new valid signature cannot be made using this key except by the hardware that is already authorized in the FPGA.

The secure storage system must also be able to store the key correctly. As discussed, if the secure boot key is leaked, a valid signature can be made that will be accepted by the secure boot system. Therefore, the secure storage system is trusted by SDSHW to be correctly implemented as well.

The anti-tamper logic must be similarly trusted, as it isolates the FPGA from the rest of this system. This provides the property of fixed isolation, as once hardware is loaded into the isolated FPGA, the rest of the system cannot observe the execution of this hardware except through the interface implemented by this hardware. As the hardware has fixed functionality, it can assume that this interface cannot be circumvented. This only holds, however, if the anti-tamper hardware is implemented correctly, and so the silicon manufacturer is also trusted to do this.

Finally, the silicon manufacturer is trusted to implement the FPGA itself correctly, so that there is no backdoor or other vulnerability in the FPGA silicon that an attacker can exploit. SDSHW relies on the trust that a bitstream loaded into the FPGA will execute the exact functionality that it was designed to do. If an attacker can exploit a vulnerability in the FPGA to force the hardware to perform a different function, then fixed functionality is lost.

These trust relationships can be summarized as: *the silicon manufacturer must implement the FPGA correctly and faithfully*. This is the same trust model as in secure hardware, as any application must trust that is was manufactured correctly. SDSHW uses fixed secure hardware to protect the FPGA, and the properties of this hardware are extended to the reprogrammable hardware that is loaded into the FPGA. Assuming that this assumption holds, SDSHW can maintain the same threat model for the initial hardware locked to the FPGA during self-provisioning.

SDSHW introduces a new threat vector, however, in the form of the update system. To maintain the same properties as secure hardware, this update system must not be exploitable to allow for an attacker to load an unauthorized bitstream into the FPGA. The definition of "authorized," however, varies depending on the security policy implemented by the secure hardware update system itself. If an attacker can meet this security policy, then they can cause hardware to be loaded, as it will be authorized. Therefore, applications must decide which security policy to choose and are trusted

to implement it correctly. For example, if hardware requires both user authorization and an update signed by the developer, the application is trusted to prompt the user correctly for this input and determine the correct type (*e.g.,* input of a PIN, pressing of a trusted input, or even a combination of both) and that the signature verification is performed correctly and cannot be forged. This update mechanism makes a choice to export some process trust to the developer, but still mediates this trust by requiring the update to be verified by the hardware being updated. So long as this update system is implemented correctly, fixed functionality can be maintained, as the hardware will always be executing the functionality it was designed to perform; if updated, this occurs using this functionality and is an extension of the hardware's properties.

SDSHW does not make any attempt to defend against threats that secure hardware is vulnerable to. FPGAs are not inherently more resistant to advanced physical adversaries and 'decapping' attacks that other silicon systems. In an FPGA, such attacks would likely need to be performed while the FPGA is still powered to directly observe the state of the running hardware, but attackers could potentially attack the secure storage to discover state instead. Power attacks, such as glitching attacks and power analysis, are also possible with FPGAs, just as they are possible with silicon. In short, SDSHW provides the same threat model as secure silicon hardware, but does not provide any additional protections. That is not the goal of this dissertation; we merely wish to prove that using SDSHW is as secure as secure hardware, not more secure.

Furthermore, SDSHW does not defend against the applications themselves being implemented incorrectly. Just as in secure silicon, an SDSHW application must not accidentally expose its data or allow its execution to be hijacked. SDSHW does not provide a framework in which secure hardware applications gain increased security, but only provides a platform for them to be implemented. Applications are still responsible for not compromising their own security, just as they are in a silicon application.

4.4 SDSHW Platform Implementation

To illustrate the capabilities of Software Defined Secure Hardware, we implemented our platform on a modern FPGA and built several proof-of-concept applications. Specifically, we used the Xilinx ZCU102 Evaluation Board [1], which consists of a Xilinx Zynq UltraScale+ FPGA that couples a quad-core ARM Cortex A53 CPU with Xilinx's programmable logic fabric as separate subsystems on the same SoC. In this section, we illustrate how we implemented the three core aspects of our platform (self-provisioning, secure storage, and a secure update) on this FPGA. The FPGA contains secure fixed hardware that is largely intended for intellectual property protection and anti-tampering defenses, which makes it compatible for our requirements. In the sections to follow, we describe two secure hardware applications we built on top of this platform: a secure filesystem protected by hardware encryption similar to Apple's Secure Enclave (Section 4.5) and an isolated software execution environment with remote attestation similar to Intel's SGX (Section 4.6).

Our complete implementation consists of the self-provisioning process, which initializes the FPGA to provide its security properties and authorize initial hardware, the secure storage system, to hold the secure boot keys, and the secure update system, which is implemented as another application in the initial hardware. The applications from Sections 4.5 and 4.6 are applications implemented in two separate bitstreams, which also include the needed secure storage logic to support these applications an the secure update system. These bitstreams are provisioned separately into our SoC for each application during its evaluation, meaning that we perform the the self-provisioning for each application on the SoC, using the bitstream for each application individually for each evalutation. In the following sections, we present how the individual components of SDSHW are implemented, which are common to each of the applications presented in Sections 4.5 and 4.6.

[1] https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

4.4.1 Self-Provisioning

The self-provisioning system of SDSHW is needed to establish the FPGA into a secure state, and as such, allows SDSHW to provide its security properties. The first step to implementing SDSHW is to provide this system. This implementation must interface with the secure hardware proved by the FPGA to configure the secure boot system correctly and to set the correct anti-tamper configuration.

The SoC we are using contains secure boot hardware, but this hardware can only be configured from the coupled CPU. The same can be said for the SoC's anti-tamper hardware- it provides the capabilities we need, but also is only accessible to the CPU. This means that the self-provisioning system must be implemented as a small provisioning operating system that runs on the SoC before it has been configured.

As our SoC is a Xilinx product, it uses a proprietary standard for loading software and bit-streams into the device. The boot flows follows a series of steps, which are executed by a fixed bootloader, known as the Boot ROM. This bootloader exists as a small amount of code in a one-time writeable memory that is programmed when the SoC is in Xilinx's production facilities, and afterwards cannot be changed. The SoC CPU is hard-wired to load this code at power-on, which then performs a number of actions:

1. The Boot ROM is loaded and examines the boot mode from jumper settings. Our implementation always assumes these are set to boot off an SD card.

2. in SD card boot, the Boot ROM loads the file 'boot.bin' off the SD card's first partition. This file is in Xilinx's proprietary format.

3. The Boot ROM examines system to see if authentication or decryption of the boot.bin file is required. On the first boot of self-provisioning, this will not be required. After provisioning, this will be required.

4. If no authentication is required, the Boot ROM loads the first partition of the boot.bin file, the First Stage Bootloader (FSBL). The FSBL then loads other partitions from the boot.bin

file, including the FPGA bitstream and operating system. The FSBL programs the bitstream to the FPGA and loads the operating system into memory.

5. If authentication is required, the Boot ROM must first derypt the FSBL. The Boot ROM reads the public key from the boot.bin file, and uses a fixed Keccak-384 accelerator to compute the hash of the RSA-4096 public key. The resulting hash is compared to a hash stored in the eFUSE array (one-time programmable configuration registers). Assuming that the hash matches, the Boot ROM then uses the public key from the boot.bin file to verify signatures of the file's partitions (*e.g.,* FSBL, bitstream and operating system). The Boot ROM loads the FSBL if these signatures pass, and then performs the previous step.

Secure boot in our SoC is therefore implemented as RSA-4096 signatures of Keccak-384 hashes. During self-provisioning, the provisioning OS will be loaded without authentication as a specially-prepared boot.bin file, using Xilinx's tools. The OS will be provided the initial bitstream, FSBL and OS that will boot in future power cycles, and will generate the device-specific secure boot RSA-4096 keypair. The OS will then generate a new, signed boot.bin using these keys and the initial hardware, FSBL and OS, using Xilinx's tools to generate the correct proprietary format.

The self-provisioning OS will then perform a Keccak-384 hash of the public key and store it into the secure boot eFuse configuration registers; it stores the complete keypair into secure storage (the OS also initializes the secure storage, as is described later). The self-provisioning system does not directly perform any system configuration of anti-tamper logic in our SoC, as these settings are lost on reboot. Instead, these are done with the FSBL that is included in the signed FSBL, meaning that the system provisioner must have selected or designed this FSBL to perform the correct actions.

For our device, the FSBL disables the processors ability to reprogram the FPGA (through the processor configuration access port), disables JTAG access, and locks out access to the system interconnect access control logic from the CPU. This is performed on each boot, and is enforced

by the SoC's secure boot system. Thus, the secure boot system effectively isolates the FPGA from the CPU and prevents the CPU from reconfiguring the system to get access back.[2]

After the OS finishes these tasks, the SoC is provisioned correctly and can boot in future to launch the initial secure hardware. As all boot will be required to be signed, and only one compatible boot.bin file was ever generated, only this hardware can ever exist, assuming the keys are never leaked from the secure storage.

4.4.2 Secure Storage

As mentioned in the previous section, the self-provisioning operating system has to initialize the secure storage system on our SoC. This is because our SoC does not have any persistent storage connected to the FPGA. Therefore, we had to implement a separate set of steps to make secure storage work.

The secure storage system is implemented as two parts: a subsystem in the FPGA secure hardware logic is implemented by the developer to encrypt data before it leaves the FPGA, and a proxy agent running on the CPU (*e.g.,* in the operating system) receives encrypted data and writes it to persistent storage, or reads it back and provides it to the FPGA. This means that during self-provisioning, the OS must launch the initial hardware and run the secure boot keypair through this subsystem in the FPGA in order for it to be encrypted such that the FPGA can later decrypt it.

We recognize that this method of secure storage is effective for storing data that never changes (*e.g.,* the secure boot keypair), but is not sufficient for storing secure hardware application state. For many FPGA applications, replay-resistant secure storage is needed. To provide this, we make use of an AES accelerator that exists in our SoC to be used for bitstream encryption, but can also be used by the CPU for general-purpose encryption. This is shown in Figure 4.3.

This AES accelerator the SoC to encrypt or decrypt using a key stored in a special secure storage implemented as battery-backed RAM (BBRAM). The BBRAM is implemented to be write-

[2]This process is highly customized to our SoC. Other systems may have completely different access control systems or connections between the the FPGA and CPU. Even in Xilinx systems, such as Zynq7000 devices, the precursor to our SoC, the access control configuration is completely different.

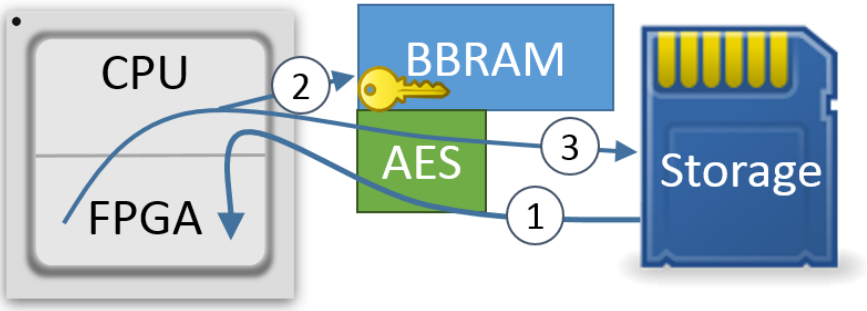

Figure 4.3: **Secure Storage** — (1) A trusted CPU agent uses the AES-GCM accelerator to decrypt the storage from the SD card, with the encryption key stored in BBRAM. The FPGA is given the decrypted value to read or modify. Modifications are given back to the CPU. The CPU generates a new AES key, overwriting the old key in the BBRAM (2) and encrypts the data under this new key, writing the ciphertext to the SD card (3). Any attempts to rollback or replay old storage data will not succeed, as old storage can no longer be decrypted.

only to the CPU, meaning the AES accelerator is the only device able to read the key from BBRAM after it has been written. The self-provisioning system initializes the BBRAM with a generated key during provisioning. The CPU has the ability to use this hardware for arbitrary encryption, but as the data that is encrypted by the FPGA is already encrypted, the AES accelerator is never able to actually decrypt the data.

Instead, we use that fact that the BBRAM can be reprogrammed to implement replay protection. This is the only secure storage medium on the SoC that can be rewritten – there is secure storage in a limited amount of eFuse memory for storing keys, but this memory cannot be rewritten. To implement replay protection, a new key is generated on each boot by the trusted FSBL and programmed to the BBRAM after the current storage is decrypted and provided tot he FPGA. The storage proxy on the CPU then simply runs any data from the FPGA through this accelerator and stores the results on the persistent storage (SD card).

As the AES accelerator implements AES GCM, any replay of storage encrypted under an old key will be detected and rejected. The FSBL makes use of this to provide the replay-protection. At each boot, the FSBL loads a file holding the encrypted secure storage and decrypt it with the accelerator, and provides it to the FPGA. The FSBL then generates a new key and encrypts the the decrypted data under this key using the accelerator (the AES accelerator can be toggled to encrypt

using an arbitrary key or the BBRAM key) and stores the new file, along with the old one, to the SD card. Finally, the FSBL programs the new key to the BBRAM, overwriting the old key.

This sequence is intended to prevent a situation where power is lost when reprogramming the BBRAM key. If power is lost at any point, either the old data or the new data will still be decryptable, as the BBRAM will either have the old or the new key. This assumes that writing the key to BBRAM is atomic, and requires the FPGA to wait before attempting to write data to the secure storage until this sequence has been completed. Therefore, the FSBL will also provide a signal to the FPGA to indicate when complete and the secure storage is available for writing, via the storage proxy.

The primary risk to this system is denial-of-service by the operating system. The OS may refuse to execute the proxy, preventing the FPGA from writing to secure storage, or may remove the use of the AES accelerator by replacing the proxy with a different implementation. Unfortunately, these risks cannot be avoided without the FPGA having direct access to storage. The risk can be mitigated by increasing the collateral damage of such an attack. This can be achieved by having the FSBL interact with the FPGA to increment a counter in the secure storage after the replay-protection sequence has occurred, and then provide a measurement of the secure storage, and allow the FPGA to store this as well, before the FPGA interacts with the untrusted proxy. This allows the FPGA to detect a DoS attack, as the FPGA will be able to tell if the hash of the storage is the same between boots, which indicates that the proxy is not writing the FPGA's storage when the FPGA requests it. In this case, the FPGA can refuse to execute, and even can shut down the system, if configured with that capability.

On the FPGA side, the secure storage is implemented as a simple microcontroller, using the Microblaze soft CPU, that receives encrypted data from the CPU and decrypts it, placing the data into isolated buffers for each subsystem in the FPGA that needs access to it. This microcontroller listens for write requests from each of these applications and writes the data back to the CPU on these requests (accomplished by watching signals from the applications in these buffers in the FPGA). This microcontroller also performs the hash checks on the secure storage by interacting

with the FSBL on startup, and is wired with the ability to put the FPGA into reset if it detects malicious behavior by the untrusted proxy.

4.4.3 Secure Update System

The final component of SDSHW is the update system. Although we are unable to prescribe a single implementation, as that would require selecting a single security policy for all users of SDSHW and implementing a system that relies on features of a certain FPGA, we do provide an example system that works on our SoC and should be reasonable for many applications. The security policy we select is developer authorization via an ED25519 signature over the update bitstream and local user authorization through the input of a PIN from the operating system, and the toggling of a button connected directly to the FPGA.

The update system is implemented as a subsystem in the SDSWH platform that runs each of the example applications described in the next sections. This subsystem is connected to the secure storage microcontroller with a separate buffer, and only this buffer is allowed to access the secure boot key, as hard-coded into the microcontroller's operation. The subsystem is also connected to the CPU and exposes an API to allow for a potential update to be loaded. The subsystem has a hard-coded ED25519 public key of the developer (*i.e.,* our public key, as we are the developer).

The security policy for updates requires that any updates be signed by this public key, a user must input a PIN, and a physical button must be pressed. This PIN is not present until a user sets it, and is set by the first user that takes ownership of the device. The update system is wired to the reset system of the FPGA as well as the secure storage, and holds the rest of the secure hardware in reset if a PIN is not set. A PIN can be specified through the same API exposed to the CPU, and once a PIN is first written, the update system stores it in the secure storage and allows the FPA to execute.

During self-provisioning, the self-provisioning OS also signed a special update OS that is booted by the FSBL in this situation, and is signed by the secure boot keypair. When an update is specified, the CPU sets a flag indicating an update request and reboots. The FSBL sees

this flag and boots the trusted update OS, rather than the untrusted normal OS. The update OS provides a PIN attempt, the update, and an ED25519 signature over the update to the hardware update subsystem.

If the signature checks pass and the PIN attempt was correct, the update system toggles an LED and the user pushes a button as the final update authorization. If too many PIN attempts are made (*e.g.,* we set five attempts as this threshold), the update system refuses to accept the update during this power cycle and records the number of PIN attempts made. The system then requires a certain number of power cycles before attempts are again accepted (this is to emulate anti-hammering systems used by TPMs).

Assuming all the authorization steps are performed correctly, the update subsystem provides the secure boot key to the trusted update OS, which then uses the Xilinx tools to sign the new update and generate a new signed boot.bin file. The update OS then attempts to securely wipe the key from memory, clears the update flag, and reboots. Now the new, updated hardware can be executed in future boots.

The main vulnerability in this system is the need to expose the key outside of the FPGA so that it can be used by Xilinx's tools. Unfortunately, so long as the format of the boot.bin file is proprietary, this will be required, as a compatible file cannot otherwise be generated. This leaves the secure boot key vulnerable to memory attacks, such as a cold-boot attack, where the key could theoretically be extracted from memory, even though the update operating system is trusted and does not execute any external code or open any network connections. This risk is reduced due to the fact that we require the user to provide physical authorization, but can be exploited if a valid update is provided, the user's PIN is compromised, and the attacker has physical access to the system (a user could theoretically prevent a cold boot attack if they are legitimately applying an update, so long as their device is not stolen during the process). Unfortunately, this is a limitation imposed by Xilinx's implementation of the device and cannot be completely solved [3].

[3] We could implement a simple Linux OS in the FPGA on a Microblaze CPU to use the Xilinx tools, but the needed resources would likely be too much for our FPGA, if other applications are to be supported.

4.5 Secure Filesystem

As a first secure hardware application that builds on our platform, we implemented an encrypted filesystem that provides functionality similar to Apple's Secure Enclave. In this application, an amount of data is protected by hardware from access outside the device by encrypting it using a key that is stored in the secure storage that cannot be exported. An AES encryption engine in the secure hardware application is the only component that can read this key and the untrusted operating system passes data to be encrypted or decrypted when it is accessed. In addition, the encryption engine will not perform any operations until a PIN has been provided correctly within a certain number of attempts. The operating system will provide the PIN to the hardware to unlock the key, and only then can data be decrypted.

Comparison to the Secure Enclave Similar to Apple's solution, the PIN is intended to prevent access to the encrypted data without requiring a long password from a user. Apple provides a similar solution in its iPhones that prevents access to a device and its storage after a certain number of PIN attempts have been reached. Apple's Secure Enclave technology enforces this by encrypting all storage with a key that is held by a secure storage system and accessible by a secure coprocessor running trusted code. This code will not boot or decrypt the system if the PIN attempt is exceeded. Our system attempts to emulate this functionality by not unlocking the storage without a successful PIN entry, and then locking out further attempts. We make use of the secure storage system of the FPGA to keep track of PIN attempts, in addition to securely storing the key.

PIN Initialization To initialize the secure hardware application when it is first used, a user will provide an initial PIN that will be stored in the secure storage (*i.e.,* if no PIN is in the secure storage, the first one received will be stored). This implements a TOFU (trust-on-first-use) security model, which should be sufficient, as no sensitive information can be stored using the system if the PIN is not set (the hardware will not perform encryption without a PIN entry). To transfer to a new user, the system can be reset using the update mechanism in factory reset mode, as described below.

Update and Reset To make use of the update capabilities that are provided by our platform, we also designed an update mechanism in the trusted hardware that has certain application-specific requirements, which can also be used to factory reset the system (*i.e.,* clear the encrypted storage and allow for a new PIN to be set). In order to allow new hardware to be authorized to run in the FPGA, the hardware requires the currently set PIN to be entered correctly. However, in the cases where the PIN needs to be reset, the hardware will also accept a new PIN, but will first generate a new encryption key and delete the previous one, effectively deleting the data that was protected by this key. In this case, a new PIN can be set, and then can be used to update the hardware. Once the PIN has been set correctly, the hardware will release the secure boot key to sign the new hardware configuration. As this is part of the secure hardware application, these requirements also cannot be circumvented.

Access from the Operating System To interface with this system, we also designed a FUSE (File Systems in Userspace) filesystem that encrypts and decrypts its contents by passing them through the hardware encryption engine after a successful PIN has been provided. We use the Python FUSE bindings to implement it, and present a performance benchmark of the filesystem in Section 4.7.

4.6 Secure Coprocessor with Remote Attestation

For the second example, we implemented a secure hardware application that provides similar features to Intel's SGX secure trusted execution environment, namely hardware-enforced software isolation and remote attestation (illustrated in Figure 4.4). The hardware is centered on the use of Xilinx's MicroBlaze soft processor [119] to realize the isolated software execution environment, and we developed software tools to allow applications to be built in a similar manner to the SGX software development kit (SDK). In this section we elaborate on the hardware design, the software development kit to develop software applications, and an example software application (password manager) that was built with our software development kit.

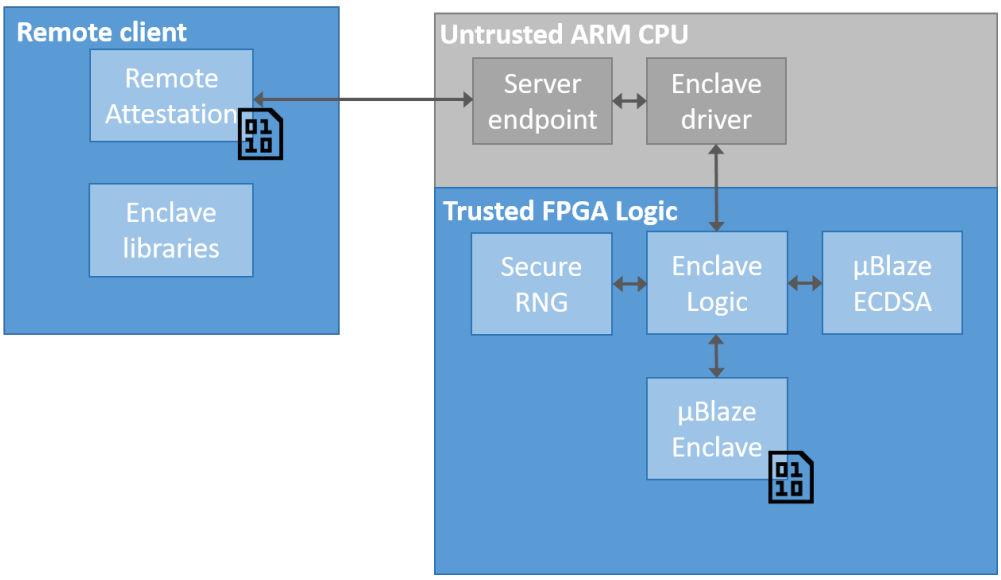

Figure 4.4: **Secure Coprocessor and Remote Attestation Design** Here we run the FPGA as a coprocessor and are able to enforce software isolation and perform remote attestation. A remote attestation client uploads a program to an untrusted server. The program is launched in a MicroBlaze CPU in the FPGA by trusted logic, which also signs the program code and performs a key exchange. The driver communicates with the program in the MicroBlaze over a shared buffer and relays data to the client.

4.6.1 Hardware Design

Isolated Software Execution Environment

To provide software isolation and remote attestation, we implemented a MicroBlaze soft CPU inside the FPGA as part of the secure hardware application. Any code that executes in this CPU is isolated from the untrusted operating system and can be trusted to execute once loaded. In order to securely program this CPU, we utilize custom logic that ensures that any trusted code (*i.e.,* a trusted "enclave" program, similar to SGX) is loaded, and that a hash of this program and a signature are performed. This "enclave logic" accepts binaries to run in the isolated CPU (*i.e.,* the enclave CPU) and programs the enclave CPU memory directly, while simultaneously calculating a SHA512 hash of the program. Only this logic has direct access to the enclave CPU's memory, so the only way to change the program is to overwrite it with a new program.

In addition, this logic reads an ECDSA private key from the secure storage, and uses this key to sign the hash of the signature and a message from the enclave CPU during the remote attestation

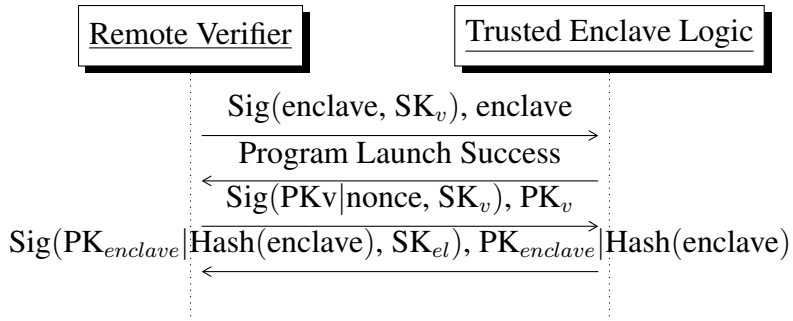

Figure 4.5: **Remote Attestation Sequence**: In the remote attestation protocol, the remote verifier uploads a program (enclave) signed by its private key (SK_v). The enclave launches the program and notifies the verifier, which then requests an attestation by sending its signed public key (PK_v). The enclave logic uses this key to derive a shared secret for the enclave and responds with a signature of an ephemeral public key for the enclave ($PK_{enclave}$) and the hash of the enclave, signed by a long-term key for the enclave logic (SK_{el}).

process. As shown in Figure 4.4, a remote client can upload a program to services running in the untrusted operating system, which will then pass the program to the enclave logic. This logic will then launch the trusted code on the MicroBlaze CPU in the FPGA. An untrusted program (*i.e.,* the "Enclave driver") can interact with this code through the enclave logic using a special shared memory buffer that is designated for this purpose.

Remote Attestation

The attestation protocol implemented by our secure hardware and companion software is shown in Figure 4.5. In this protocol, a remote verifier uploads a program signed by its Ed25519 private key (SK_v) [120]). The program will be launched by the enclave logic, and the verifier will be notified upon completion. The verifier will request an attestation by uploading its signed public key (PK_v). The enclave logic generates an ephemeral key pair for this attestation to establish a shared secret for the enclave ($PK_{enclave}$, $SK_{enclave}$), and signs $PK_{enclave}$ and the hash of the enclave program with its long-term attestation key (PK_{el}, SK_{el}). The enclave sends these to the verifier, along with a certificate chain configured at provision time by the root of trust for this device. Using this certificate, the verifier then verifies the signature and checks that the hash matches the expected hash of the uploaded enclave program. If so, the verifier can calculate a shared secret us-

ing $PK_{enclave}$ and SK_v, just as the enclave logic calculates a shared secret using PK_v and $SK_{enclave}$. Using this shared secret known only to the verifier and the isolated enclave, a secure channel can be established.

Random Number Generation

To generate secure ephemeral keys during this process, we have included a cryptographic random number generator within the trusted hardware of the FPGA, as implemented by the Cryptech OpenHSM project [121]. The module draws randomness from both the LSB of A/D conversion noise as well as a ring of digital oscillators implemented as a set of adders with the carry-out inverted and fed back as carry in. This entropy is collected and hashed using SHA512 to whiten the numerical randomness and remove any bias introduced by the entropy sources. The resulting digest is used to seed a ChaCha stream cipher by providing the key and IV. A counter is maintained such that the stream cipher can be reseeded when reaching a maximum number of blocks. Random numbers are provided as a 32 bit value, which are sampled multiple times by the enclave logic to generate secure keypairs.

4.6.2 SDK

In addition to designing the hardware of our software isolation system, we have also designed a software development kit to make it easier to develop software applications that run in the system. Figure 4.6 shows the major components of the SDK. A developer creates untrusted code that runs on the ARM CPU of our system in the untrusted operating system (arm.c), code that implements the trusted functions that are run in the isolated MicroBlaze CPU (mb.c), and description of the API the application wishes to use to communicate between the trusted and untrusted code (interface.json). This interface describes the inputs and outputs of the trusted code as well as the function signatures of the specific methods. The developer also has access to the enclave library (encl.h, encl.c) that provides functions to launch an enclave, which is done by interacting with the enclave logic.

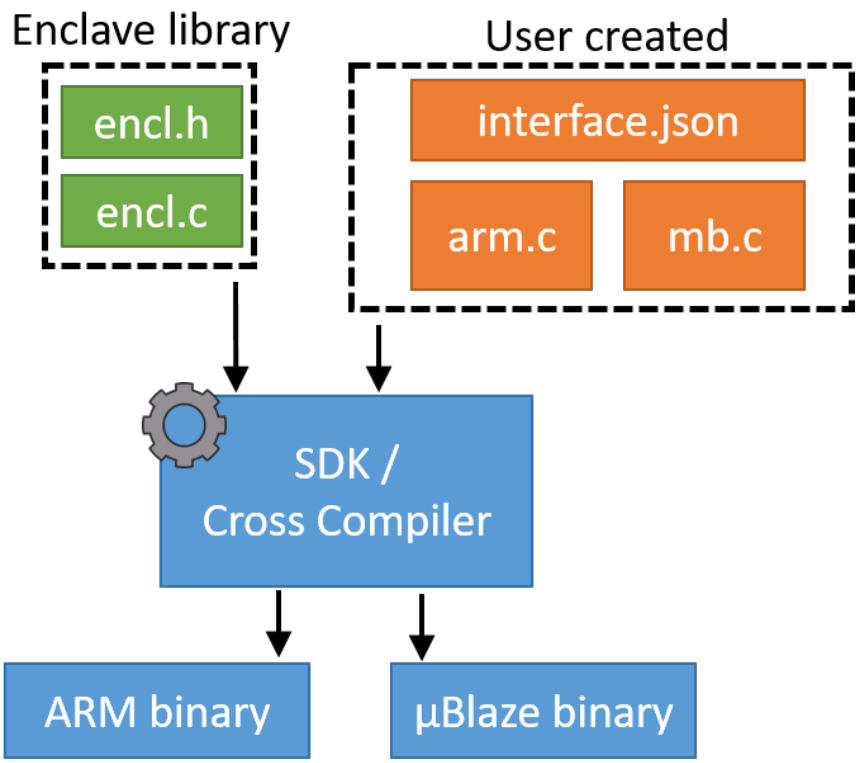

Figure 4.6: **SDK Development Flow** The enclave library (green) provides functions the software developer needs to launch an enclave and perform remote attestation. The developer writes their code (orange) that they wish to run on the ARM CPU and the isolated MicroBlaze CPU, while calling the desired functions in the enclave library. The enclave library and user created code are fed into the SDK (blue), which cross compiles the code and produces executable binaries for both the ARM CPU and the MicroBlaze CPU.

The developer provides their code to the SDK, which will use the API interface definition to generate communication code between the MicroBlaze and the ARM CPUs using the dedicated shared buffer. It cross-compiles the code for the ARM and MicroBlaze instruction sets respectively, producing two binaries. The (untrusted) ARM binary loads the trusted MicroBlaze into the enclave using the enclave library.

4.6.3 Password Manager Application

As an illustration of running isolated software in this secure hardware module, we implemented a password manager that encrypts stored credentials under a master password. Passwords are encrypted and decrypted in an enclave with only the encrypted data being stored in persistent

storage. To access a password, the enclave must be provided with the encrypted data and a master password. The enclave then derives a decryption key using this password and a device-only key that can only be accessed from the enclave, as it is stored in the FPGA's secure storage.

To use the manager, a user provides their master password to a client program which interacts with the enclave. The user has the option to enter information for passwords, usernames, and identifiers (*e.g.,* a website). This information is given to the enclave to encrypt and passed back to the client application to store in persistent storage. Retrieving data is acheived by interacting with the client program and requesting data by its identifier, which will cause the enclave to decrypt it and return it to the client. This password manager is similar in design to an example application SGX provided by Intel [122].

Our implementation cannot remove all possible attack vectors, as the password manager must still function to provide data in plaintext in order for it to be useful for users to interact with unmodified programs. However, we can force any attacks to be *online*, in the sense that the adversary must query the password manager in the trusted enclave, rather than simply be able to make copies and reveal the entire database. This is because the encrypted password database can only be decrypted using the user's master password and the FPGA's device-only key. Even if the database is exported and the user's password is compromised, the data cannot be decrypted without interacting with the enclave running on the device on which it was first encrypted. We present a performance analysis of user interaction with the password manager in Section 4.7.

4.7 Evaluation

To understand the performance impacts of using secure hardware applications implemented using SDSHW, we evaluate the example applications that we described previously using several benchmark experiments. For all our applications, we continue to use the ZCU102 Evaluation Kit. The goal of these experiments is not to improve upon performance compared to their traditional implementations, but to show what the performance trade-off is for a developer to use the more

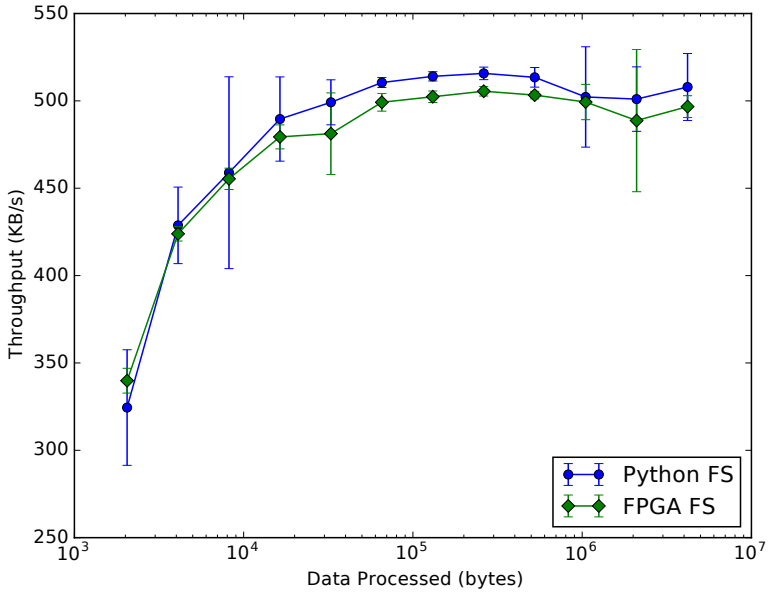

Figure 4.7: **Filesystem Performance** Write performance of a pure-Python filesystem vs. SDSHW (FPGA) filesystem. Despite being unoptimized, the FPGA implementation still only has an average overhead of 1.38%.

flexible platform of SDSHW versus using an existing platform, or even implementing a new version in silicon.

4.7.1 Secure Filesystem

To test the performance of our secure filesystem, we compared our Python FUSE secure hardware filesystem with a pure Python implementation executed on the coupled ARM CPU of our device. In both implementations of our experiment, all file system operations are implemented in Python other than the data encryption calls. For the pure-Python version, a software implementation of AES-CTR encrypts all file data, while the trusted hardware version uses a naive AES encryption system that performs the block cipher on a 16-byte input, with the CTR mode-of-operation implemented on top of this cipher in software.

We generated and stored files from 1 KB to 4 MB in a directory mounted with each filesystem and measured the throughput in writing these files to the encrypted storage directory. Figure 4.7

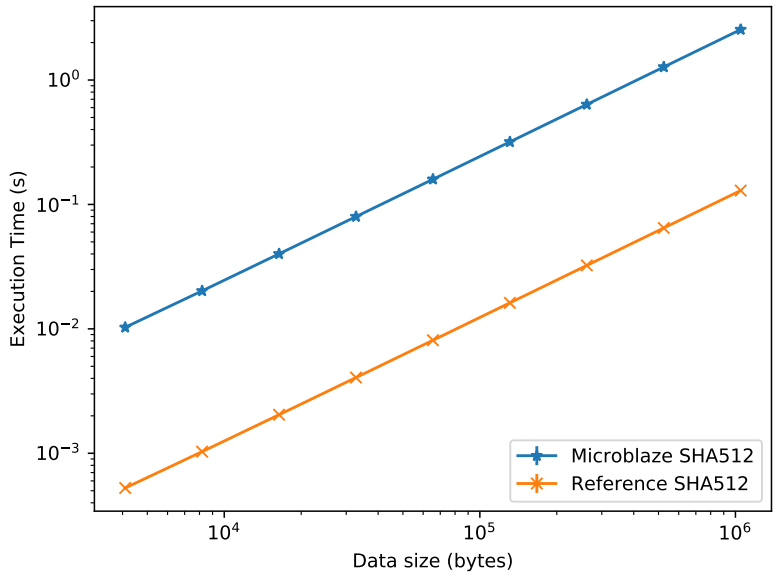

Figure 4.8: **SHA512 Enclave Performance** Performance of SHA512 hashes in an enclave running in an embedded Microblaze CPU and a reference implemntation directly on an ARM CPU. The Microblaze performance is approximately 20x slower than the reference, as expected by their relative clock cycles.

shows the throughput achieved for each written file plotted against the size of each file. Compared to the pure-Python filesystem, our naive hardware-backed filesystem offers a modest overhead of less than 2%. We expect that with modest optimizations (e.g., using DMA and other bulk transfer), our FPGA-based secure filesystem could outperform software-based implementations, as is done in Chapter 3.

4.7.2 Enclave Performance Benchmarks

To test the performance impact of executing code on a Microblaze CPU, we designed several microbenchmarks to test memory and computation performance, along with end-to-end performance tests. This is to show the performance impact of using the Microblaze CPU and whether this impact is large enough to impact even low throughput, user-interactive applications. Our experiments show that the Microblaze imposes an expected performance decrease for various computational loads, but the SDK we use to develop applications and the method of data passing between

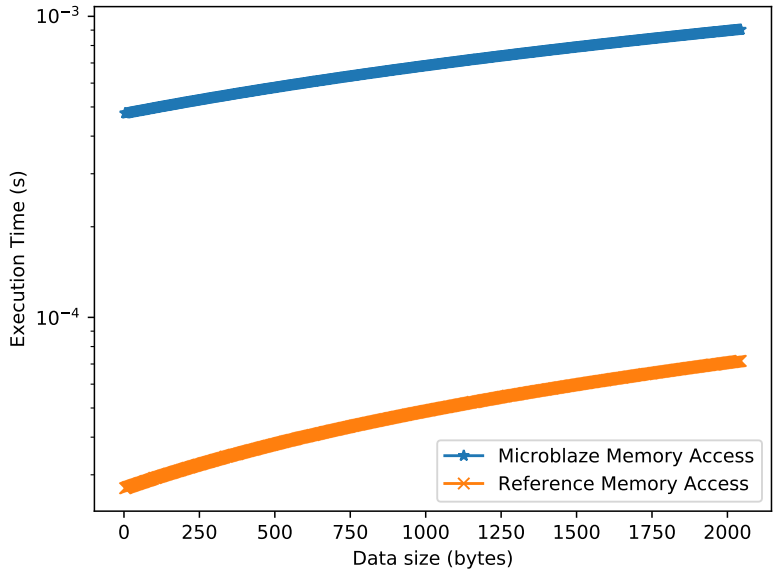

Figure 4.9: **Enclave Memory Access Performance** Memory access throughput of the enclave compared to the ARM CPU. For small memory transactions, the MicroBlaze is approximately 100x slower than the CPU, but at larger transfers, it is only 12x slower.

software and the Microblaze enclaves is not impacted to affect how performance scales. The Microblaze execution scales at the same rate as the ARM software implementation, but it has a much lower overall performance, as is expected.

Enclave SHA512 Performance We created a program that hashes a buffer of random data using SHA512 in both an enclave and directly on the main CPU. As the enclave executes on the embedded Microblaze CPU, we expect the performance to be much worse, and this experiment is intended to determine if using our SDK to create enclave programs imposes additional overhead, and what exactly the performance degradation is.

As shown in Figure 4.8, the performance of the Microblaze enclave is approximately 20x worse than the reference implementation on the ARM CPU. However, both implementations scale linearly with the size of the data being hashed. There does not appear to be any overhead caused by using our SDK to develop a program for the enclave, and it appears that the execution performance of the Microblaze CPU is the main performance bottleneck, as expected. We stress that while

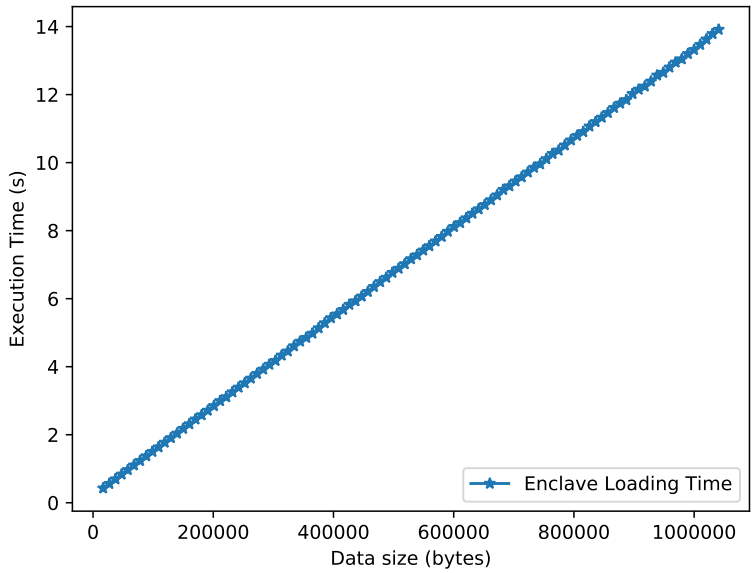

Figure 4.10: **Enclave Loading Performance** Performance of loading programs of various sizes into the enclave. The throughput of loading programs is constant at approximately 35 KB/s.

our system has significantly less performance than that of pure hardware implementations, very few secure applications require the full performance of the main processor, but instead emphasize security, isolation, and ease of implementation over raw throughput.

Enclave Memory Access Performance To measure the memory access performance of an enclave, the enclave is simply tasked with copying an input buffer to an output buffer, and the performance is compared to the ARM CPU's performance at the same task. As shown in Figure 4.9, we measured an overhead for Microblaze access times ranging linearly from 100x for small chunks of data (0-250 bytes) to 12x for larger chunks (2 Kbytes and larger).

Enclave Loading Performance Our final benchmark measures the throughput of loading enclave program binaries of various sizes. After testing using binaries ranging in size from 20 KB to 1 MB, the throughput remained constant at 35 KB/s, as is shown in Figure 4.10.

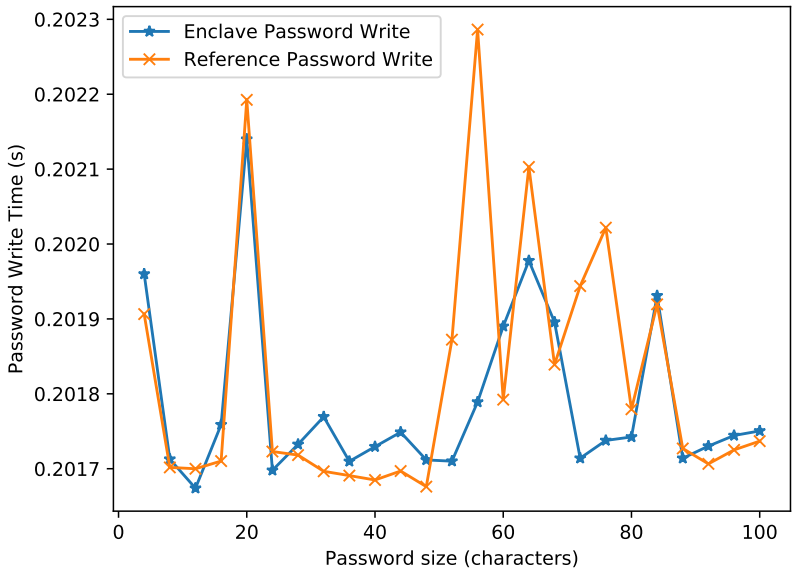

Figure 4.11: **Password Manager Write Performance** Time spent adding passwords to the password manager when protected by an enclave and when using a reference implementation running completely on the ARM CPU without an enclave.

Password Manager Performance Illustrating the point that the performance impact of our implementation commonly would impact a relatively small fraction of the overall perceived performance, we measured the time to add and retrieve passwords from the password manager application described in Section 4.6.3, for passwords of up to 100 characters in length. As seen in Figures 4.11, both with and without running in an enclave results in an average 202ms latency for encrypting new passwords (with less than 0.3 difference in the worst case). Likewise, for reasonable passwords up to 100 characters, the latency for decrypting a password from the manager is roughly 120ms for both implementations, well within the realm of usability (for passwords much larger than that, the impact of the performance difference does become noticeable as more time is spent in the enclave), as shown in Figure 4.12.

Remote Attestation Performance To measure the end-to-end performance of performing a remote attestation, we implemented a private set intersection calculation program that calculates the intersection of two sets of integers in an enclave, with one set being uploaded in encrypted form

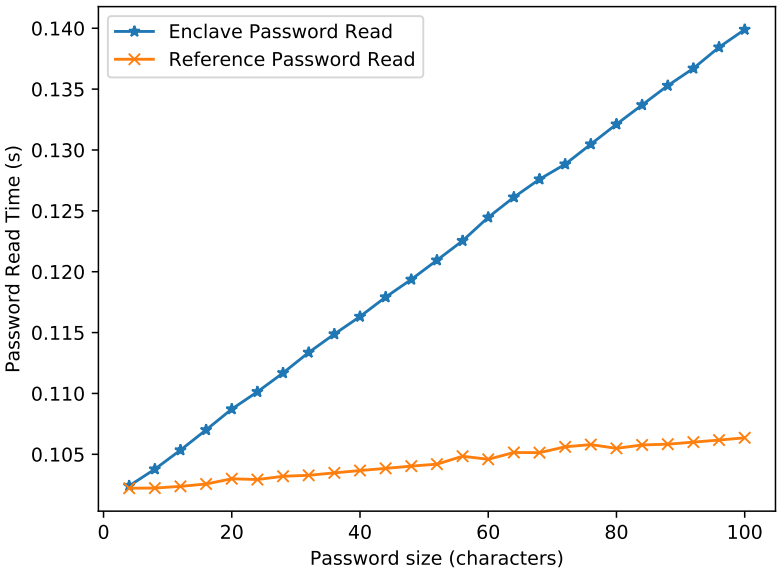

Figure 4.12: **Password Manager Read Performance** Time spent reading passwords from the password manager using enclave-protected code (top curve) and a reference implementation running on the ARM CPU (bottom curve).

using the shared secret negotiated by the remote attestation protocol, and the other provided to the enclave by the local host, similar to the contact discovery feature used by Signal [123]. In each attestation, a fixed amount of data is passed in each message, which is the public key of the verifier in one message, and then the signed public key and hash of the enclave in the response. This experiment measures the average time to pass these messages, for the enclave logic to generate the keys and sign the message, and the time for the client to verify the response and calculate the shared secret. After performing 1000 trials in ideal laboratory network conditions between a verifier and the device running the trusted enclave logic, the average remote attestation time was 107.2 ms with a standard deviation of 8.604 ms.

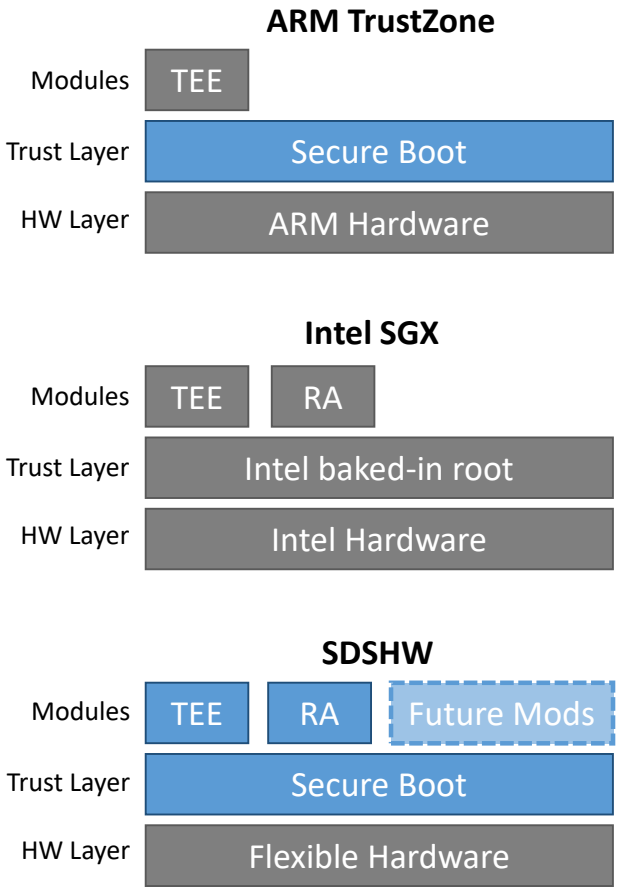

Figure 4.13: **Secure Hardware Layers** We conceptualize secure hardware into three layers, and depict ARM TrustZone, Intel SGX, and our own SDSHW platform. The hardware layer details who is responsible for the fixed hardware components/silicon, the trust layer describes who controls the root of trust for the device, and modules shows implemented features on the platform. Blocks in blue show flexible components that can be updated, while grey depicts fixed components: for example, ARM TrustZone has a flexible root of trust that can be provisioned independent of the device, while Intel SGX's root of trust is fixed in hardware. SDSHW also allows future features to be implemented after device manufacture.

4.8 Discussion

4.8.1 Trust Anchors

In traditional secure hardware systems, the root of trust is often simply the hardware manufacturer. However, in the SDSHW platform, the root of trust is configurable, as shown in Figure 4.13. This affords a great deal of flexibility, but raises questions surrounding who or what these trust anchors would be in practice. For instance, in a remote attestation setting, the server could easily

configure themselves to be their own root of trust. A concrete example would be Google configuring their own servers to use a Google-rooted certificate to verify remote attestation proofs. However, this may defeat the purpose of the remote attestation: if one trusts Google to not craft malicious attestation proofs, why not trust them to simply run what they claim (without proof) to run?

This example motivates the use of third-party trust anchors, independent from the parties that use or even build the device. These trust anchors could be trusted by a large set of users or organizations, and could either provision devices themselves, or could cross-validate other parties' root of trust certificates, allowing them to act as intermediaries verified by the trust anchor, similar to how certificate authorities (CA) operate in the X.509 PKI.

In the remote attestation example, a third party trust anchor (e.g. Verisign) may audit and review a secure hardware implementation written by Google, and sign certificates that allow that specific configuration to be used in a remote attestation feature. While Verisign must be involved in the provisioning step, thereafter, users that trust Verisign can trust that Google-provided remote attestation proofs are valid and come from a configuration vetted by Verisign.

Another possible trust anchor could be a cloud-based data center such as Amazon EC2. In this scenario, Amazon could act as their own trust anchor, renting out access to their machines to other companies. These companies in turn could use secure hardware features implemented and rooted in trust from Amazon, ensuring that the company renting the VM was protected from the data handled by this secure hardware.

4.8.2 Ideal Hardware Support

We have implemented the SDSHW platform using commodity hardware, but there may be additional fixed hardware that could simplify or improve support of SDSHW-like flexible secure hardware. In this section, we explore subtle architectural modifications that could improve, simplify, or further SDSHW support.

Dedicated FPGA storage. In our implementation, we used a combination of BBRAM and a small kernel of trusted software to load a key into the FPGA, allowing it to later encrypt writes to and decrypt reads from a system storage without needing to trust the CPU. A more elegant solution could allow the FPGA to directly write to its own persistent storage that is not accessible from the CPU.

FPGA Secure Boot. Our secure boot only supported booting trusted code into the main CPU, which in turn could program the FPGA. This required a small amount of trusted code that would program the FPGA, and then remove the CPU's access to reprogram or introspect on the FPGA before booting the untrusted OS on the CPU. An alternative, more elegant design however, could allow the secure boot to directly program the FPGA, obviating the need for any trusted code to run on the CPU.

FPGA control of CPU. As a further extension, the FPGA could have control over the CPU, rather than vice versa. For example, the FPGA could be given control over the TLB, cache, and ring level of the CPU, allowing it to halt the CPU and decide what code it should be running and from what permission level. This would allow the FPGA to take advantage of the full power of the CPU, running enclave code on it while keeping it isolated from the untrusted operating system and clearing caches or encrypting memory before swapping the untrusted OS back in.

Chapter 5

User Space Secure Hardware

The vision for this dissertation is to allow applications to dynamically execute secure hardware modules, *i.e.,* to let software use programmable hardware as a user space resource for defining secure hardware. We have proposed using FPGAs to provide this capability, as an FPGA can implement any digital circuit, and so any secure hardware system that can be implemented in silicon can also be supported by our system. Here, we describe how to bring these two solutions together to realize our vision of secure hardware as a user space definable resource.

5.1 Introduction

As we have stated in this dissertation, secure hardware enables applications that otherwise cannot be implemented based on its security properties. However, because these applications are implemented as fixed silicon, it is difficult for developers to find a platform that implements all of the systems that they would like to have. In Chapter 3, we showed that applications can actually include their own hardware definitions that can be executed in an FPGA dynamically. In Chapter 4, we showed that developers can have the secure hardware features they want in an FPGA by showing how the FPGA itself can be used securely.

However, Chapter 4 requires that all hardware in the FPGA be trusted, meaning that it cannot be shared between dynamic modules proposed by Chapter 3. This means that software cannot use

the FPGA dynamically for secure hardware as we would prefer. The need for this capability is the primary goal of this work and would enable a new class of applications. Essentially, any application that needs to protect state from the rest of the system in the face of an untrusted operator, potential threat, or risk of compromise could make use of this capability.

For example, applications that need to store sensitive financial information can store and operate on it using the secure hardware. This can be used for mobile payment systems on smartphones and for processing off-chain cryptocurrency systems. Applications can also choose to protect arbitrary data from the rest of the system in the case of device theft, such as a file sharing application (*e.g.,* Dropbox). Such an application would only store encrypted data that can only be decrypted by the secure hardware, and would be unlocked periodically by the user. In the case of device theft, even if the entire operating system is compromised, such as by compulsion by a state-level actor, the data stored by this application would be safe so long as the user does not unlock it, without requiring a custom hardware system to be implemented for a device to support it. Encrypted messaging applications (*e.g.,* Signal, Telegram) could benefit from this capability as well, as it would allow them to ensure that data remains private even if the entire software system colludes against it. Trusted computing systems such as cloud-focused TEEs like SGX can benefit from this system too. Instead of requiring a general computational solution, developers could design their own hardware that is optimized for their application and does not rely on a single hardware implementation. This enables SGX-like applications to be built with fewer compromises, such as an alternative implementation of the contact discovery application implemented by Signal using SGX.

In this chapter we implement a proof-of-concept user space secure hardware module to demonstrate the potential of this system. This module is an implementation of a private-set intersection computation, similar to Signal's contact discovery SGX service. However, there are some challenges to making our system work using the technology presented in previous chapters. We present these challenges here and describe how our system overcomes them in the next section. Our actual implementation of the system and proof-of-concept applications and their evaluation follows.

5.2 Challenges

Unfortunately, the Cloud RTR and SDSHW systems that were described in previous chapters are not compatible with each other. SDSHW makes some assumptions and imposes requirements on the FPGA configuration that Cloud RTR violates. If these challenges could be solved, then Cloud RTR can implement the SDSHW platform in a device that is properly provisioned.

Cloud RTR Challenges: Cloud RTR's primary incompatibility with SDSHW is that it requires the CPU to have complete control over the hardware that exists in the FPGA. SDSHW requires that all reconfiguration and debug access into the FPGA be disabled so that secure hardware cannot be overridden or introspected. This means that Cloud RTR cannot be implemented in a system that has been configured for SDSHW, as these capabilities will not be available, and so Cloud RTR will not be able to load new hardware into the FPGA even if a compatible bitstream is booted using SDSHW.

SDSHW Challenges: SDSHW makes an assumption that all of the hardware that is booted into the FPGA is provided by the same entity, and so the entire FPGA is trusted to have access to sensitive data. SDSHW does not provide any means directly for access control, such as to protect data in the secure storage; these capabilities are left for different implementations of the platform to provide if they are needed. The notion of loading hardware modules from different developers dynamically violates this principle, as there is no way to ensure that these modules do not access restricted resources maliciously.

5.3 Secure Slot Architecture

To overcome these challenges, we have identified four modifications that need to be made to Cloud RTR and SDSHW that make them compatible with each other. These changes are to be made to a Cloud RTR static design that executes in an FPGA that has been securely provisioned

using SDSHW. The goal of these changes is to make a system that is compatible with Cloud RTR reconfigurable hardware modules (*e.g.,* has dedicated reconfigurable slots) while not using capabilities or introducing vulnerabilities that weaken the properties of SDSHW. The changes we make are concerned with how reconfigurable modules are loaded and the connections these modules have to the rest of the static hardware in the FPGA.

5.3.1 Internal Reconfiguration

The primary incompatibility between Cloud RTR and SDSHW is the method for reconfiguring the hardware in the FPGA. In Cloud RTR, the operating system on a coupled CPU has complete control over the hardware running in the FPGA, as it has access to dedicated reconfiguration logic. SDSHW disables these features as part of the anti-tamper hardware requirements, as such a capability would allow for a malicious operating system to override any secure hardware in the FPGA without detection, and would thereby negate the security properties provided by the platform. However, we note that Cloud RTR does not require the ability to reconfigure the entire FPGA, only the actual reconfigurable slots using partial reconfiguration. In addition, all modern FPGAs include internal configuration logic, sometimes called the internal configuration access port (ICAP). We leverage the ICAP to implement a subsystem in the Cloud RTR static design that receives reconfigurable modules to load from the CPU and loads them into the FPGA directly.

The main challenge to implementing this system is that the data used to reconfigure the FPGA cannot be examined to determine where in the FPGA it is being placed, *e.g.,* if the bitstream to load is a partial or full bitstream. This is because the format of this data is proprietary, and so cannot be parsed. When loading the data into the FPGA, the logic must not be tricked into loading a bitstream that overwrites the static secure hardware; only a bitstream that is intended to go into a particular slot should be loaded. Fortunately, Cloud RTR solves this problem by requiring the cloud compiler to sign metadata for each bitstream indicating which slot it is compiled for, in addition to the bitstream itself. The loading logic in the FPGA must verify the signature over the bitstream and the metadata before attempting a reconfiguration.

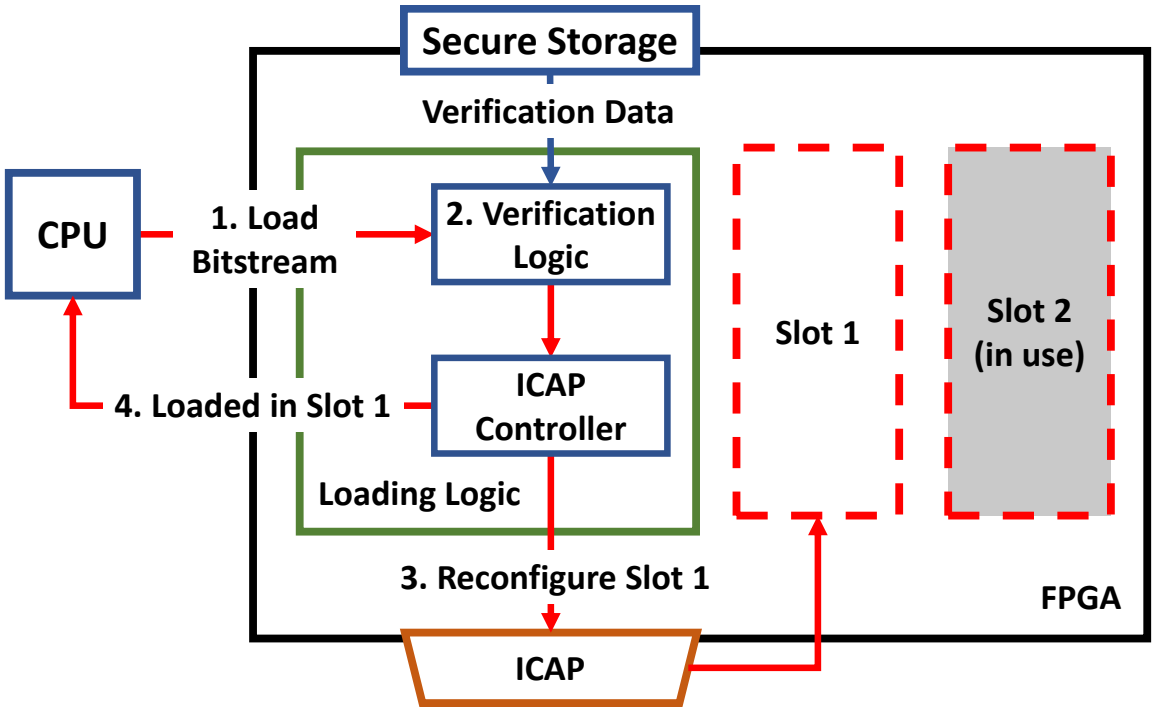

Figure 5.1: **Secure Slot Loading** The secure slot loading system is a module (outline in green) that has a sub-function that verifies bitstreams, metadata, and authorization data (such as checking a signature or a PIN) before allowing a reconfiguration to take place. The provided bitstream is then given to the ICAP controller logic if authorization is successful, and then the slot that was reconfigured is given back to the CPU.

This logic must also maintain state about the currently executing hardware modules, as the operating system is untrusted when running the SDSHW platform. The CPU may request that a module be loaded into a slot that is already occupied by a different module when a different slot is available, or request that a module be loaded into a more privileged slot. To prevent these situations, the loading logic will maintain this state and will only load modules into slots they are authorized to use, and not overwrite a slot if a different slot is available. Rate-limiting of slot overwriting is also needed to prevent hardware modules from being maliciously removed, but the limit needs to be calibrated for different device use cases. Once measured, it can be configured into the loading logic, and the SDSHW update flow can be used to update it, if needed.

Based on these requirements, the loading logic follows this operational flow, as shown in Figure 5.1:

1. The operating system requests a module to be loaded by writing to the loading logic's con-

trol registers, which are memory-mapped into the CPU's address space using FPGA-CPU interconnects.

2. The loading logic will verify the cloud compiler's signatures over the module bitstream and metadata.

3. Assuming the signature check passes, the loading logic will determine if the slot indicated by the metadata is available.

   (a) If the slot is available, it will load the module immediately.

   (b) If the slot is not available, and there is not a rate-limit constraint that prevents loading, the logic will overwrite the slot with the new module.

4. If a load was made, the logic will return a descriptor to the CPU of the slot by writing to a CPU-memory mapped output register.

To further prevent abuse of slot reconfiguring, the loading system will need to implement additional access controls. Cloud RTR introduced the concept of the privileged slot, which has greater access to system resources such as direct CPU memory access. These capabilities are useful for secure hardware applications, but are also targets for malicious modules. Therefore, at the very least, the loading logic will perform another signature check over the module against a whitelist of developer public keys that are trusted not to abuse these capabilities. This whitelist can be hard-coded into the logic and updated using the SDSHW update flow. In addition, the logic can require further authentication, such as user input of pushing a dedicated physical button or inputting an authorization code.

5.3.2 Slot Isolation

Cloud RTR slots also do not make any claims about the security of the system. They offer the ability to change a part of the FPGA at runtime, but not to do wo without compromising other hardware or observing data. It has not been proven that FPGA hardware can be designed to

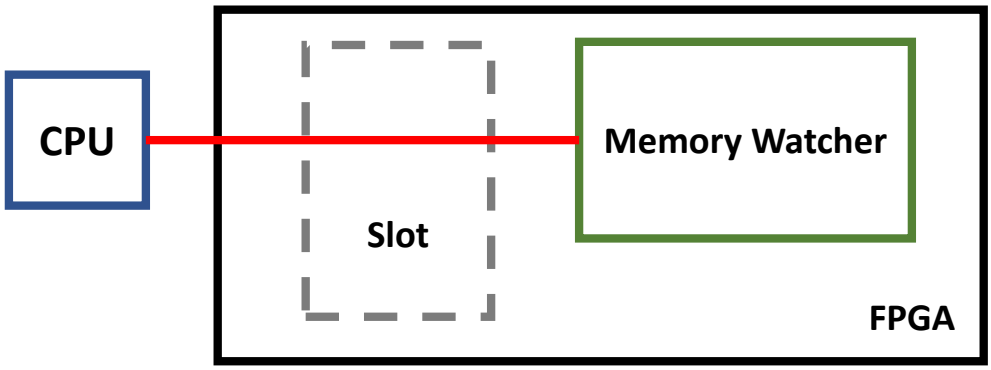

Figure 5.2: **Potential Slot Wire Snooping** If a static application in the FPGA, such as a memory watcher that scans for malware, is connected to the CPU through a wire that runs through a slot, there is a theoretical potential for the wire to be snooped. Although existing compilation tool do not allow for hardware be designed to connect to a wire in this way, it has not been proven to be impossible.

interfere with other hardware without having direct circuitry designed to perform it, but there is a risk of a reconfigurable module having a wire that carries sensitive data being routed through the reconfigurable region, and in theory, it may be possible to interfere with it, with an example shown in Figure 5.2. In addition, memory corruption attacks such as Rowhammer [124] have not been studied in FPGAs, but since FPGAs are comprised of similar memory technology to Rowhammer-vulnerable systems, this attack also may be possible. To prevent these theoretical attacks, efforts should be made in the design of the FPGA layout to prevent slots being placed near logic that is sensitive to prevent memory corruption or wire snooping, as is recommended by the Xilinx security guide [125].

This can be accomplished using two steps. When a design for an FPGA has been synthesized, sections of the FPGA can be physically reserved for certain logic. This process is known as "floorplanning" and is done before routing of wires is performed (a simple example is shown in Figure 5.3). A simple first step is to isolate the reconfigurable slots and any sensitive static logic into physically separate regions of the FPGA in such a manner that routing will not place wires through the slots (*e.g.,* having these regions on opposite sides of the physical FPGA). This initial step may be sufficient, and can be manually examined after routing has been performed. As the routing of the static design does not change when reconfigurable modules are compiled, this step

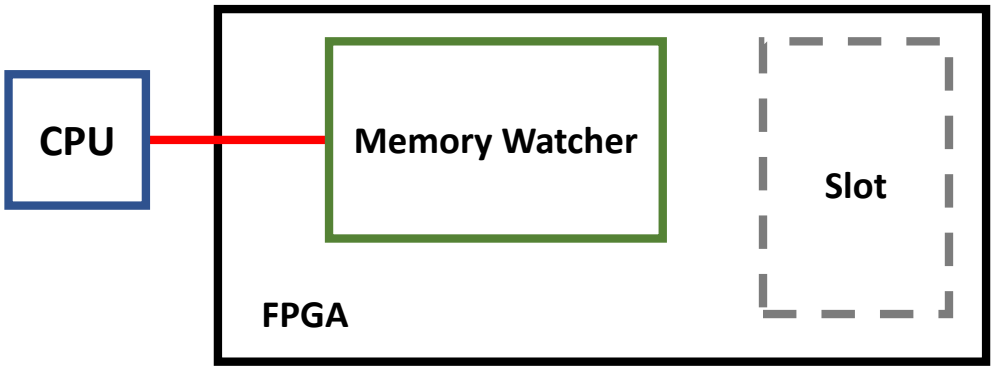

Figure 5.3: **Floorplanned Slot** In this design, the slot has been floorplanned to be isolated from the static logic so that sensitive wires are not routed through it. In this case, it is simple, but other that designs that have more sensitive static logic or more slots, the floorplanning and wire routing because more difficult.

only needs to be performed once per design.

If routing cannot be isolated in this way automatically, the design must be constrained to prevent the routing of sensitive data and the placement of sensitive logic near the reconfigurable modules. This process is more difficult to automate, but also only needs to be done once per static design. However, using constraints in this manner reduces some of the benefits of Cloud RTR and should only be done if absolutely necessary. Achieving isolation in this manner is left to future work.

It should be noted that a reconfigurable module should not be able to directly read a wire that is not part of its own logic. As the reconfigurable modules are designed as C code or synthesized netlists and uploaded to the cloud compiler, designers should not even be able to know that such wires exist, and any attempts to reference them should result in compilation failure.

It should also be noted that using design constraints to enforce isolation is only needed for static designs with high resource utilization that results in high routing contention. The more extra space there is in the FPGA that is not used by logic, the easier it is for wires to be routed without being placed in the reconfigurable modules.

5.3.3 Slot Preemption

As previously described, the loading logic in the secure SDSHW hardware will load reconfigurable modules into slots on request from the CPU and overwrite running modules in certain cases. Cloud RTR specifies that modules in slots should have metadata set to indicate whether they are preemptable or non-preemptable, meaning that they will lose important state if they are unloaded due to time slicing. Cloud RTR allowed for this, but also recommended limiting the number of non-preemptable modules running simultaneously, as the number of available slots in a given device is limited. This is further complicated by secure hardware applications, where preemption may impose a danger to the system.

To address this, we require that hardware modules still include this prememption flag in their metadata, but we also add signals to the module to indicate that a reconfiguration is going to occur. When a module must be unloaded, the loading logic will assert a signal and wait a certain number of cycles based on a maximum time indicated in the metadata that the module needs to store state. The module in turn will make use of the SDSHW secure storage system to store this state. Once a finished signal from the module is asserted or the timeout is reached, the running module will be overwritten with the new module. This is to balance the need for secure hardware to be running with the challenge of limited FPGA resources. All slots will be updated to include these signals, and modules are responsible for implementing the needed state-saving logic and indicating the time needed to do this, if they use this functionality. Modules also can still indicate that they are non-preemptable, but again, the number of these modules running will need to be limited to conserve resources.

5.3.4 Secure Storage Access

As previously described, the reconfigurable modules should have access to the SDSHW secure storage system in order to safely store application state. This is needed to support many secure hardware applications, *e.g.,* secure cryptography and TPMs that generate and store state. In order for reconfigurable modules to have the same capability, the secure storage must be accessible.

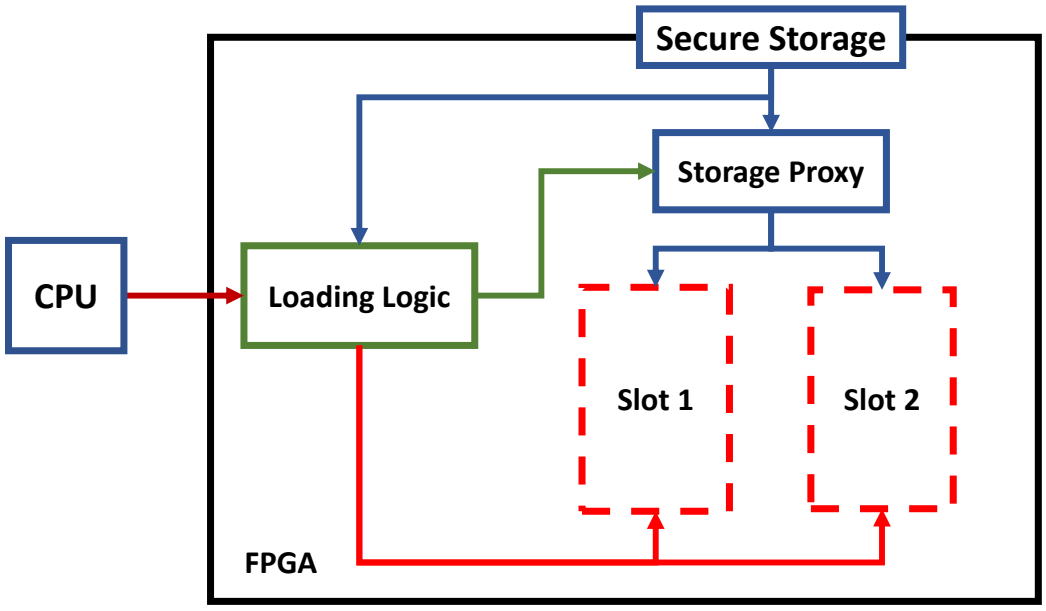

Figure 5.4: **Secure Storage Proxy** So that the arbitrary hardware that can be loaded into a slot cannot read all data out of the secure storage, such as the secure boot keypair or data used by other applications, a proxy is placed between the slots and the storage in the FPGA. Since the slots can only change the hardware in the slot area, but not the interface to the rest of the FPGA, this is the only way to access the storage. The storage ensures that the hardware in each slot can only access data that it has generated, based on information on the currently running hardware from the loading logic. Static hardware in the FPGA can be directly wired to the secure storage, as this hardware is present at design time and all comes from trusted sources.

The secure storage system could be connected directly to the slot definitions to provide this access, but this imposes a security vulnerability. Since SDSHW assumes that all hardware in the FPGA is trusted, there are no access controls imposed on the secure storage, as any hardware in the FPGA is trusted to access it. This assumption fails when hardware is loaded from unknown developers, as unrestricted access to the secure storage would allow for access to the SDSHW secure boot key, and would then allow it to be exfiltrated and used to take over the device.

Instead, we implement a proxy system between the secure storage and the slots definitions that processes read and write requests to the secure storage and only allows a module to access data that it has generated. This is only used for access to storage from the slots; any other hardware in the static secure hardware can be connected to the secure storage, since all this hardware is from the same trusted source. This is shown in Figure 5.4.

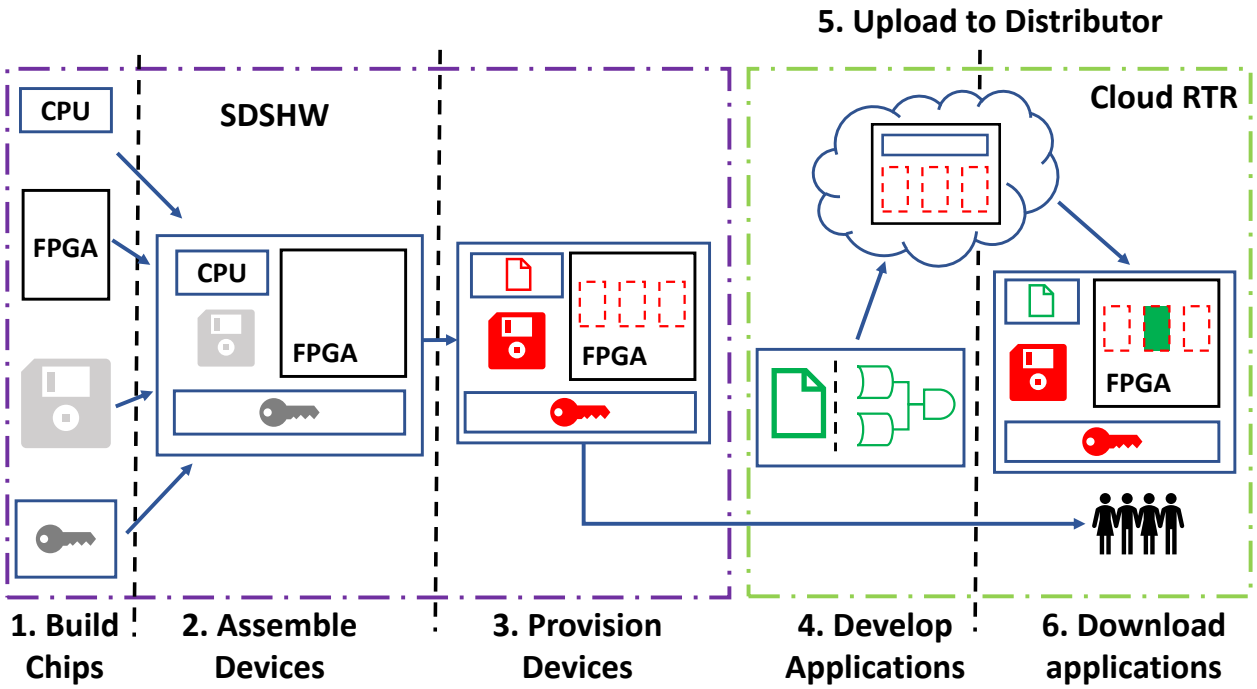

Figure 5.5: **User Space Secure Hardware Overview** User space secure hardware combines Cloud RTR and SDSHW. Based on the six roles defined in Chapter 2, each party performs a action in the device creation flow. In SDSH, the silicon manufacturer builds chips, the hardware assembler builds devices from these chips and the system provisioner provisions hardware into the device. In Cloud RTR, applications developers upload apps and hardware to the distributor and users download applications to the device that the system provisioner provisioned with SDSHW.

This system will need to know which module is running in each slot, and so will receive this data when a slot is configured. The loading system will generate an identifier for each module, such as its hash value, and specify this identifier and the slot that it is running in to the storage proxy. The proxy then uses this identifier to lookup where in the secure storage to process read and write requests. Any requests from a given slot will be assumed to come from the module identifier that the loading logic placed in that slot; the proxy will receive a request from a slot, look up the module identifier for the slot in its internal state, and use that identifier to look up the modules's data in storage.

5.3.5 Combining Solutions

With these solutions, we have made Cloud RTR compatible with SDSHW. To provide our complete system of user space secure hardware, we require that an FPGA be provisioned with SDSHW and for a Cloud RTR bitstream to be executed in the provisioned FPGA. We expect that chips are provided by silicon manufacturers that hardware assemblers can combine into a system that provides the FPGA and fixed hardware requirements of SDSHW, which are already available as off-the-shelf systems today. We then expect a system provisioner to have the FPGA self-provision into a secure state that executes a modified Cloud RTR bitstream that incorporates the solutions we have described above. Finally, application developers can design software and hardware modules to target the Cloud RTR reconfigurable slots in the bitstream using the Cloud RTR development flow, and the system's operating system will have the needed interfaces to let software load their hardware into the FPGA.

Specifically, our system is designed and used in these steps, which are outlined in Figure 5.5:

1. Silicon manufacturers provide chips with an FPGA and required fixed secure hardware.
2. Hardware assemblers take this chip and other components and assemble a device.
3. System provisioners design or select a modified Cloud RTR bitstream that incorporates our solutions.
4. System provisioners load the SDSHW self-provisioning system into the FPGA and provide the modified Cloud RTR bitstream as the initial configuration.
5. Developers design applications for these modified slots and upload them to the distributor.
6. The distributor provides these applications to users, and the users execute them. The applications load hardware into the secure slots as needed.

5.4 Implementation

To prove that the realization of user-space secure hardware is feasible and that Cloud RTR can be modified to implement the SDSHW platform, we used an existing device to demonstrate our secure slots architecture. This system is the ZCU102 Development Kit, which couples a quad-core ARMv8 CPU with a modern Xilinx FPGA. This system includes the required fixed secure and system configuration hardware that is needed by the SDSHW platform, and so is usable for our demonstration.

We implemented a modified Cloud RTR implementation that has the changes to the loading and slot definitions as previously described. The heart of this implementation is a modified version of the static design that is presented by Cloud RTR. The modifications that we made to this system were to implement loading of modules internally by the FPGA, implementing a proxy system for access-control to the secure storage and changing the definition of slots to support secure storage and privileged resource access.

5.4.1 Secure Slots

In our modified version of Cloud RTR, there are still reconfigurable slots that are used for dynamic modules, but now there are multiple classes of slots. Essentially, we have the vanilla slots of Cloud RTR that do not provide any security properties, 'secure' slots that implement the changes we have outlined in this chapter, and 'privileged' slots, which are secure slots that have access to sensitive resources, such as full CPU memory space access. We note that upgrading a vanilla Cloud RTR slot into a secure slot does not remove any functionality, so we upgrade all slots to secure slots in our implementation. Not all applications can or should need access to privileged resources, however, so we only provide a limited number of privileged slots, which are connected to the system's DRAM controller.

This slots are physically isolated in the static design and floorplanned to be separate from other logic such that sensitive wires are unlikely to be route through these slots. After performing place and route on the design, we verified that this was the case, as the other static logic outside of the slots is not complicated, and so does not required abnormal wire routing to be performed.

5.4.2 Secure Storage Proxy

As the slots need access to the secure storage, there are ports in the slot interface that modules can use for reads and writes. However, modules should not be able to access all stored data, as they are not trusted to control the FPGA. Therefore, we implement a simple proxy subsystem of the loading logic. When the logic loads a module into a slot, it provides this information to the proxy in the form of the slot number and the SHA512 hash of the module. When any module is loaded, the proxy will partition the data they store, and only allow reads or writes to the storage area corresponding to the hash. This allows for modules to still store data, but not access other data for modules or hardware in the static design. We implemented these features by simply extending the secure storage system that was implemented for SDSHW in Chapter 4.

5.4.3 Secure Loading

To be able to load reconfigurable modules without compromising the security of SDSHW, these modules need to be loaded directly by the FPGA into the correct slots. Therefore, we implemented a simple subsystem of the static secure hardware using the Microblaze soft CPU that interfaces with the FPGA's ICAP logic. The Microblaze CPU then exposes an API to the CPU to allow for modules to be queued for loading, along with metadata indicated the modules requirements (*e.g.,* is a persistent module, is a privileged module, signatures). All modules are hashed as they are loaded, and this hash and the slot the module was loaded into is passed to the storage proxy as described.

There is a security problem when loading the bitstream, however, in that the logic does not know where the bitstream for a module is destined without other information. The ICAP logic

will load it into the correct place, but the Microblaze subsystem cannot parse the bitstream, as its format is proprietary. However, Cloud RTR already requires the module and its slot information to be signed by the cloud compiler. To address this vulnerability, we verify this signature in the loading logic to ensure that the Cloud RTR system has compiled the bitstream for the slot that it is being loading into. In addition, if the module is specified to be privileged, it also must be signed by a key in a whitelisted set of keys that represent trusted module sources.

We combine all of these modifications into a single bitstream that we run on our test device. The device is provisioned with SDSHW to provide the security properties that are needed by the secure slots, and we use this bitstream for implementing applications described in our evaluation.

5.5 Evaluation

To prove that our vision of user space secure hardware is both possible and practical, we implemented a proof-of-concept application that uses our system. We then evaluate the performance of this application to determine the tradeoffs an application developer must be aware of when using the FPGA, and use it to determine the throughput for reconfiguring the FPGA using the ICAP. This application is an implementation of a real-world private set intersection service, similar to the contact discovery designed for SGX by Signal. This system allows for an encrypted list of contacts to be uploaded and compared to a database of contacts without exposing the uploaded list. In the following sections we describe how this application works in detail and present benchmarks of its performance compared to a software implementation.

5.5.1 Contact Discovery Performance

For our proof-of-concept application, we extend our private set intersection experiment from the previous chapter to implement a functional system similar to an existing service. The encrypted messaging service Signal allows users to exchange messages without the service provider having any information on the communication other than the knowledge of the total set of Signal

users [123]. However, in order to discover other users, a new user must match their set of contacts against the database of users help by Signal, which is performed by Signal in the cloud and is stored as a list of phone number hashes. To improve user privacy, Signal has implemented this contact discovery service in SGX so that users can upload their contacts in an encrypted form and have them matched against the database without directly exposing this list to Signal. This service is open-sourced, so any user can verify that the correct code is running using SGX's remote attestation system.

We have implemented a similar system that is executed in a secure reconfigurable module. An application will include this module and load it into the FPGA using the internal reconfiguration system, just as with the previous application. The application maintains two lists of contacts: one for the user's uploaded contact list and one for a portion of the database. To use the hardware, a user encrypts and uploads a list of phone number SHA512 (Signal contacts are phone numbers). These encrypted hashes are passed to the hardware, which decrypts them using a secret in the secure storage and appends them to the contact list. The software application then provides portions of the database to the hardware in batches, where the hardware matches them against all of the contacts in the list and indicates which in the batch match. This is performed until the entire database has been passed through the hardware.

We compare the performance of this hardware with an alternative implementation implemented in C++ and executed on the coupled ARM processor CPU. The results are shown in Figures 5.6 and 5.7. As can be seen, the performance of the FPGA implementation is approximately 30 times slower that the software implementation, and this relationship is maintained as the size of the database increases. However, the time to determine a match between a database item and a contact is constant, and is only approximately 20 times slower in the FPGA. This shows that the primary bottleneck to using the FPGA is passing data between the CPU and the FPGA, and is consistent to results found in Chapter 3. Results for this proof-of-concept could be improved with a more sophisticated data-passing model, as was done in that chapter. This evaluation does not attempt to make a performance argument, however. The goal is to present this tradeoff to developers, so that

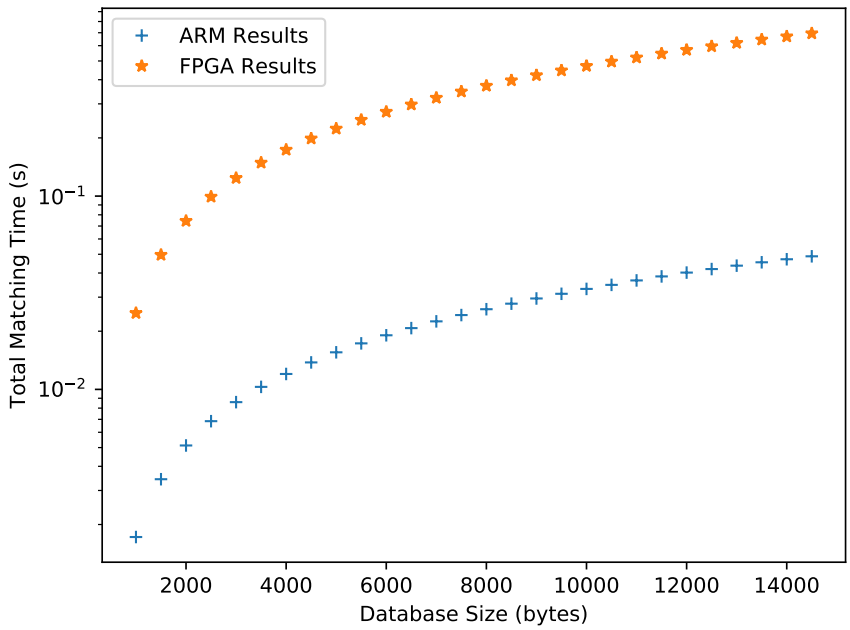

Figure 5.6: **Contact Discovert Overall Performance** The performance of the contact discovery application, including loading of the database into the FPGA. The FPGA is approximately 30x slower than the ARM CPU, but the two implementations follow the same scaling patterns as the size of the database is increased.

they are aware of the basic design tradeoffs of performance vs. programmability that are required to use our system. A more complex application design that transfers less data into the FPGA and uses more data-parallelism in the FPGA would likely achieve better results.

The results of an alternative implementation are presented in Figure 5.8. In this implementation, the database is stored in FPGA memory, and the contacts are loaded in for each user. This implementation takes advantage of more parallelism of the FPGA, but is limited by the amount of memory available in the FPGA to store the database. Because of this, our experiment was limited to a database that can have 7500 contacts at most. As shown from the data, the FPGA's performance time is initially higher, but increases at a much lower rate than the CPU. With more available FPGA memory (*e.g.,* streaming directly from system memory), or by using a different database storage design (*e.g.,* using SHA256 instead of SHA512), this memory limit can be mitigated. The performance of the FPGA would then surpass that of the ARM CPU's for larger database sizes. This would be ideal for Signal, as their database of users would likely be in the millions [126].

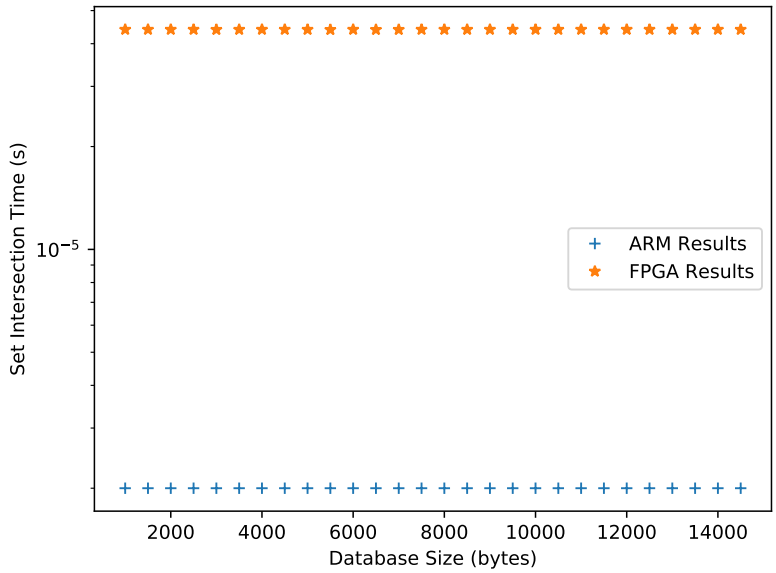

Figure 5.7: **Contact Discovery Intersection Performance** The performance of the contact discovery application when only considering the performance of the set intersection of the database and the uploaded contacts. The FPGA is approximately 20x slower than the ARM CPU, but is consistent with all database sizes, as this measures the time to check one item in the database against the uploaded contacts list.

5.5.2 ICAP Benchmark

We used these two implementations to benchmark the ICAP's reconfiguration performance. The FPGA static design that we used for these experiments contains a reconfigurable slot, and both designs for synthesized to be partial bitstreams. Using these bitstreams, we used a C++ program to write them to the ICAP. The reconfigurable slot is approximately 50% of our device's available logic, so as to provide the needed memory resources to the second contact discovery implementation.

The resulting partial bitstreams are 12 MB in size – any hardware module that is compiled for a particular slot will be the same size as any other module, as the bitstream needs to configure the entire area of the slot, no matter how much of it is actually utilized. After running our benchmark for 100 trials, we determined that the ICAP can reconfigure this area in roughly 68.2 seconds, achieving an average throughput of 173.4 KB/s with a standard deviation of 1.223 KB/s.

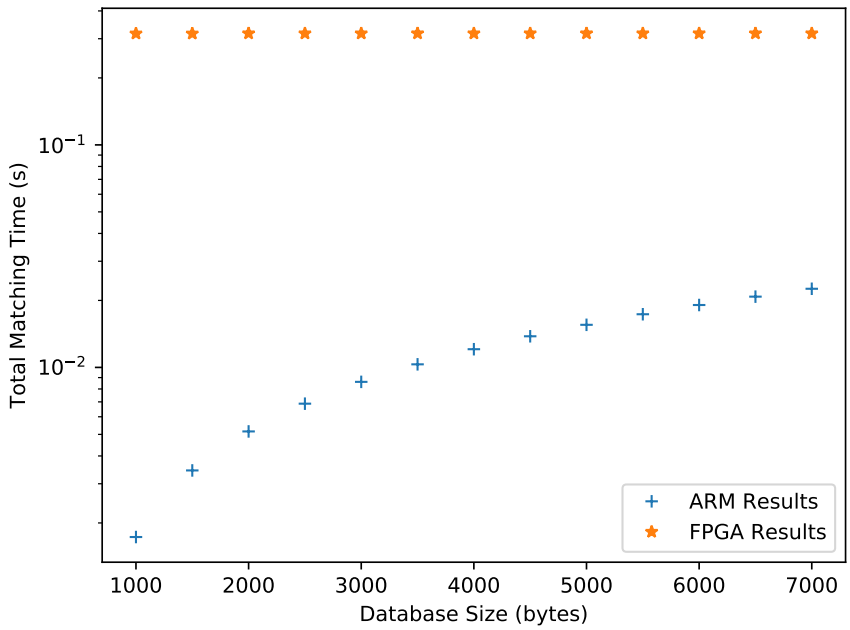

Figure 5.8: **Contact Discovery With Pre-loaded Database** The performance of the contact discovery application when the database is already loaded into the FPGA. For small database sizes, the software implementations is faster than the FGPA, but as the data size increases, the FPGA intersection time remains constant, but the software time increases.

Based on this performance, the ICAP will likely not be able to handle reconfigurations that happen too often. Slots in devices implementing user space hardware are likely to be smaller so as to increase the sharing of the FPGA, but the time slicing of the FPGA cannot achieve the rate of software at this time. For this reason, the ICAP reconfiguration logic of user space secure hardware will need to enforce more strict rate-limiting and minimum residence time for modules (*i.e.,* modules will be allowed to stay in the FPGA for at least a certain amount of time before being at risk of preemption). This does limit the total number of modules that can practically share the FPGA, but we do not expect this to be problematic. As discussed in Chapter 3, not all applications are expected to want to include their own custom hardware. We expect the same for secure hardware, and the slot system of Cloud RTR that user space secure hardware uses can only support as many concurrent applications as there are slots, which will calibrated for the most optimal size for each device by the system provisioners and application developers.

5.6 Security Analysis

The stated goals of this dissertation were to provide a means for secure hardware to be used as a software resource and to ensure that this hardware maintains the same threat model as traditional secure hardware. In this chapter, we have combined the systems from the previous chapters to meet this goal. However, we have not proven that the threat model described in Chapter 2 is maintained. In fact, there are a number of additional threat vectors that are created by our system that must be examined.

Our analysis focuses on the two properties of secure hardware that we have identified, fixed functionality and isolation. As secure hardware derives its protections against the rest of the system and physical adversaries from these properties, if we can show that we still provide these properties then we will also provide the same threat model.

5.6.1 Fixed Functionality

This property results from the fact that when an application is implemented as a physical digital circuit in silicon, the functionality of the chip cannot be practically be changed. This means that so long as the chip is powered, this functionality will perform the same as when it was manufactured. However, if the hardware can be changed, *i.e.,* if the FPGA used to execute this hardware can be reprogrammed, we lose this property, as the functionality of the original bitstream is changed.

We do not claim that FPGAs do not operate in this manner; this functionality is the exact reason we need to use an FPGA. Instead, we claim that this property can be provided to a developer by ensuring that the hardware loaded into the FPGA always provides its function while it is loaded. We argue that this claim does not need to be maintained at all times universally, and that it only matters to applications if the hardware functionality is executing when the application is expecting it to. Therefore, we claim that any hardware in the FPGA that is running using our system provides this property so long as it is still running.

For static hardware in vanilla SDSHW, this means that the hardware provides this property so

long as the bitstream is running, and if an update is provided, this means that the functionality has changed, not the property itself. These properties hold so long as the update mechanism cannot be subverted by an adversary. This means that the update authorization enforced by the hardware must be sufficient to prevent this from occurring. Requiring a signature to be verified over the update should be required at minimum, but some form of local authorization should be required, such as the input of a PIN from a user or the pressing of a physical button. This prevents the compromise of the key from compromising the secure hardware, as local authorization would still be required.

However, for secure hardware that is loaded dynamically, these same properties are more difficult to provide. This is because changing the running hardware module is the intended purpose of this system and for an application to rely on this property, the hardware must not be unloaded until the application is ready for it to be unloaded. In theory, the application designers could encode the times they expect the hardware to be running in the configuration metadata (*e.g.,* always running in the background, running when the application is running), but these conditions are difficult to verify. For example, if an application requires a module to always be running when the application is executing, the hardware can load the module on request, but cannot tell if the application is running since this is controlled by the operating system, which is untrusted. This could be solved by requiring all hardware modules to be persistent, but this greatly reduces the sharing potential of the FPGA. Instead, we present the option for authorization to be required for configuration, especially if the reconfiguration requires for a currently running application to be unloaded. This authorization request is presented to the user, and can be performed with a PIN entry, or preferably, a trusted input directly to the FPGA. For example, the internal reconfiguration logic can be updated to require such an authorization when a reconfiguration would overwrite an existing module, and a user would press a physical button on a device to approve the reconfiguration. Such physical interfaces are already manufactured into existing devices, and can easily be made to be trusted inputs for the FPGA, and we implement this as an optional step for authentication in our reconfiguration logic.

However, for a cloud situation where there is no user to press a physical button, a different solution is needed. In this case, we can implement time slicing of slots that are contended, meaning

that no single hardware application can use them exclusively. This requires more in-depth design of our reconfiguration logic and the applications themselves, but is achievable. To support this, a hardware application will provide the minimum execution time it needs for a cycle along with the preemption time required by the preemption system we described. If contention for the running slot occurs, then the application will be allowed to run for its execution cycle and then signaled to be unloaded, where it has opportunity to unload its state. After that, the slot is reprogrammed with a new hardware that has the same information.

This means that all hardware has access to the slot in the face of contention. In addition, rate limiting can be applied to contended slots to prevent a certain module from being starved. As the hardware cannot tell absolutely if an application is executing, this is the only way to ensure that a hardware module is running when the application needs it. Rate limiting is important, however, as this contention over the slots will greatly decrease the system's performance. In addition, since the hardware cannot tell if an application has exited, these hardware modules need to be keep being time-sliced until the FPGA reboots or receives a new signal. The hardware applications themselves can signal they are finished after receiving a remote message or after they have complete their task however.

These properties are enforced by the FPGA's static configuration, but this configuration can only be trusted if the secure fixed hardware has been properly configured and if this hardware does not have any vulnerabilities. The provisioning step required to establish SDSHW ensures that the system is configured properly. For our test device, this means generating a secure boot keypair, programming this key to the secure boot one-time programmable configuration registers, setting the bits in this configuration to require all boots to be signed by this key. The provisioning step then signs a single configuration that is provided as the initial hardware and boot software. This is the only software that can be booted by the device, assuming the key was generated securely and not exfiltrated during the provisioning process. This software then disables the reconfiguration and debug systems of the CPU to prevent observation and reprogramming of the FPGA. For our device, the secure boot system verifies a 4096-bit RSA signature of the booted software at each

boot, using a key that is stored as a Keccak-384 hash in its configuration (the full key is provided as part of the boot file). Assuming this cryptography cannot be broken and is performed correctly and consistently, the secure boot system itself operates as designed.

So long as the system provisioner performed these steps faithfully, fixed functionality can be provided. In essence, SDSHW and by extension, user space secure hardware, relies on the FPGA being provisioned into a secure state.

5.6.2 Fixed Isolation

The isolation properties of user space secure hardware are provided in part by disabling debug and reconfiguration access by the CPU, but also by designing the static bitstream and slots correctly. For vanilla SDSHW, simply disabling external access is all that is required for this property, as there is no other way to change the FPGA state without going through these ports connected to the CPU in our device. So long as the booted software that is loaded by secure boot disables these ports after the trusted FPGA configuration is loaded, they will never be available to compromise this property.

For user space secure hardware, however, the fact that hardware can be changed at any time meant that we had to allow for hardware that is not part of the trusted static configuration. Therefore, to maintain this property for all static hardware and all potential hardware modules, these modules must not be able to observe the rest of the FPGA. We have therefore designed the slots so that each reconfigurable region is an isolated sandbox with only certain interfaces to access secure storage and data exchange with the CPU. The assumption is that once defined in this manner, the hardware loaded into the modules cannot change the wire routing, and so is trapped. This assumption holds so long as hardware cannot be designed to 'glitch' the FPGA somehow to cause wires to be connected that were not intended. To our knowledge, this is not possible, but it has not be studied by research. We leave further exploration of this subject to future work. However, short of being able to change the FPGA's running configuration maliciously, any hardware in the reconfigurable slots will be isolated from other hardware in the FPGA, thus providing fixed isolation.

Chapter 6

Discussion, Future Work and Conclusion

In this dissertation, we have presented our vision of user space secure hardware and provided an implementation with complete example applications. Not only does our vision solve a problem facing developers today, in the fact that they cannot always have access to the secure silicon features they need, but it can be directly implemented on existing devices. Here we discuss the implications of our system and what future work needs to be done for all of the various components.

6.1 Discussion

Implications

With our system, applications are able to create their own implementations of hardware systems that do not require the development and manufacture of new silicon. Furthermore, we can provide the same security properties for this hardware as if it was implemented in silicon. This provides many new avenues for applications to be implemented on, as it not only allows for them to design their own custom accelerators, but allows for them to process private data and perform secret computation that is application-specific, and protected from the rest of the system. This means that applications can perform any computation using our system and be safe from any exploit of the operating system or even from a physical adversary that can replace arbitrary parts of the system.

The clearest benefit to consumer devices is protection from theft. For example, if our system is run on a smartphone, applications can store and process data in a secure hardware module when the application needs to use it. If the device is stolen, even if the operating system protections are bypassed, this data cannot be accessed and the computation provided by the secure hardware modules cannot be changed. This is especially useful for defending against state-level threats, as these actors, such as government and law enforcement agencies, have the ability to compel software and hardware manufactures to disable protections. By provisioning the system the way we do, there is no party external to the device that can be compelled to disable protections. State-level actors instead will need to use their compulsion powers on users of devices directly.

Assumptions and Threats

Our system also relies on both a correct provisioning process provided by the system provisioner and for the hardware itself to not have any vulnerabilities. In addition, applications need to be compiled and delivered faithfully to devices by the application distributor. We show these trust relationships in our trust model, and these are the same as current hardware. However, in our system, this trust model only needs to be maintained by each party at one time, *e.g.,* at provision time or when the application is compiled.

Some of these stages are difficult to audit, such as ensuring that the hardware was manufactured correctly or that the system was provisioned securely. However the compilation results of the cloud compiler are simple for a developer to verify so long as the static design is open source. The purpose of the cloud compiler is to compile hardware for all different platform versions for different apps and to recompile when new versions are released, but if these designs are known, any single compilation result can be independently verified. However, all of the trust relationships in the trust model are one-time relationships, even if they cannot be audited. The system is secure so long as these parties can be trusted at the time they perform their required actions.

The primary technologic threats are related to how hardware is updated and if the properties of secure hardware can be circumvented. The SDSHW platform provides a means for hardware to be updated by the developers if they chose, but if the developers implement it incorrectly or use a

weak authentication system, then an adversary can cause a malicious update to be accepted. We do not require a certain set of requirements for the implementation of an update mechanism, but we have provided suggestions to reduce the risk of the update mechanism being abused.

Essentially, this advice can be summarized as requiring authorization for an update by the party that is using the system. For personal devices, these would mean requiring both the user and the application developer to provide some sort of authorization, such as by pushing a physical button and signing the update respectively. In the case of a cloud application, the developer can implement a remote attestation system in their application that requires authorization from the developer directly before the update will be accepted. The goal of these schemes is to prevent the hardware from being updated without knowledge of the application user, as this violates the property of fixed functionality, as the hardware will be executing a function that is not what is expected by the application. Updates are allowable, but the application user needs to be aware of when updates occur in order for this property to be maintained.

6.2 Future Work

There are still features to be implemented for all three systems we present: Cloud RTR, SDSHW, and the complete user space secure hardware system. We would also like to explore more applications for each of these systems to determine different usage models.

6.2.1 Cloud RTR

There are several avenues for future work in Cloud RTR. Since we did not explore the possibility of using FPGAs to achieve power savings in devices, this topic is open to be explored. Prior research has suggested that offloading computation to the FPGA and disabling high-powered systems can reduce the overall system draw in certain situations and is worth exploring. We would also like to demonstrate the Cloud RTR system for other vendors and device usage contexts, such as Altera FPGAs and use cases in data centers.

6.2.2 SDSHW

There are several avenues for future work for SDSHW as well. We implemented several example applications, but we did not implement all of the possible secure hardware features that we identified. Implementing these features would make SDSHW more practical for developers, as developers could use them directly as libraries. In addition, there are a number of hardware enhancements that we identified in Chapter 4 that we would like to have added to future devices. We would also like to interact with silicon manufacturers to see if these features could be added to augment the fixed hardware in FPGA systems so that implementing the SDSHW platform is less complex. This includes allowing the FPGA direct access to the secure hardware, having a dedicated storage device accessible only to the FPGA, and possibly even making the FPGA the master of the system to allow for the coupled CPU to be used for secure computations.

6.2.3 User Space Secure Hardware

Besides the future work that is available for the Cloud RTR and SDSHW components, there are several avenues that can be pursued for the overall vision. More applications beyond a proof-of-concept should be explored to further explore its potential. Work also needs to be done to determine how much interference a reconfigurable module can cause on the rest of a running FPGA bitstream, or if it can read private data. If such interference is impossible, some of the isolation requirements on reconfigurable slots we proposed will not be necessary. However, if it is possible, the amount of isolation that is needed between different FPGA modules will need to be determined and our isolation efforts will be updated with this information.

We would also like to explore better models for application-hardware communication. In the current implementation of user space secure hardware, applications have a strong assumption that the correct hardware is running and are given a a communication channel directly with it by the operating system. However, the hardware has no guarantee that the software that originally loaded the hardware is still running in the operating system. If this hardware provides some sort of secure function for an application, it may be desirable to ensure that *only* this application can receive

the output. Currently, there is no way for the FPGA to determine what software is running on the CPU. We would like to explore a possible method to establish such a secure channel. Currently, the operating system is trusted to provide it. It may be possible, however, for the FPGA to verify the operating system that is running, and based on this, either have access to a list of running software or some other way to verify the application that is interacting with it.

6.3 Conclusion

We have proven our vision of user space secure hardware in this dissertation. Our Cloud RTR and SDSHW systems provide the building blocks to implement user space secure hardware, namely by levering cloud compilation to allow for applications to provide their own hardware and self-provisioning to ensure that an FPGA is securely configured. By combining these solutions, we are able to achieve our vision by having the FPGA take control of configuration, thus making Cloud RTR compatible with SDSHW.

The resulting solution, user space secure hardware, demonstrates that our vision is possible. Furthermore, we demonstrate it using existing hardware, meaning that it can be practically realized today. We have enabled developers to design their own secure hardware and access it from their software applications as a user space resource when run on a provisioned device.

References

[1] *iOS Security - iOS 11*. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[2] Fan Zhang et al. "Town Crier: An Authenticated Data Feed for Smart Contracts". In: *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Vienna, Austria, 2016.

[3] Joshua Lind et al. "Teechain: Scalable Blockchain Payments using Trusted Execution Environments". In: *CoRR* abs/1707.05454 (2017). arXiv: 1707.05454. URL: http://arxiv.org/abs/1707.05454.

[4] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *ACM Trans. Comput. Syst.* 33.3 (2015).

[5] *Amazon EC2 F1 Instances: Run Customizable FPGAs in the AWS Cloud*. https://aws.amazon.com/ec2/instance-types/f1/.

[6] Mehrdad Majzoobi, Farinaz Koushanfar, and Srinivas Devadas. "FPGA PUF using programmable delay lines". In: *Information Forensics and Security (WIFS), 2010 IEEE International Workshop on*. IEEE. 2010, pp. 1–6.

[7] *Project Catapult*. https://www.microsoft.com/en-us/research/project/project-catapult/.

[8] A. M. Caulfield et al. "A cloud-scale acceleration architecture". In: *l IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016.

[9] Andrew Putnam et al. "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services". In: *Proc. Annual International Symposium on Computer Architecuture (ISCA)*. 2014.

[10] *CES: Intel GOes for self-driving cars*. https://www.electronicsweekly.com/news/design/ces-intel-goes-self-driving-cars-2017-01/.

[11] *Zynq-7000 All Programmable SoC*. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/. URL: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/.

[12] *Zynq UltraScale+ MPSoC*. http://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[13] Victor Costan and Srinivas Devadas. *Intel sgx explained*. Tech. rep. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086.

[14] *Secure Golden Key Boot*. https://rol.im/securegoldenkeyboot/.

[15] *CVE-2016-3287*. Available from MITRE, CVE-ID CVE-2016-3287. July 2016. URL: https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3287.

[16] *CVE-2016-3320*. Available from MITRE, CVE-ID CVE-2016-3320. Aug. 2016. URL: https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3320.

[17] *Altera SoCs*. https://www.altera.com/products/soc/overview.html. URL: https://www.altera.com/products/soc/overview.html.

[18] *Vivado High-Level Synthesis*. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/. URL: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/.

[19] Paramvir Bahl et al. "White Space Networking with Wi-Fi like Connectivity". In: *Proc. SIGCOMM*. Aug. 2009.

[20] Nick L. Petroni Jr. et al. "Copilot - a coprocessor-based kernel runtime integrity monitor". In: *Proc. USENIX Security Symposium*. San Diego, CA, 2004.

[21] Jad Naous et al. "NetFPGA: Reusable Router Architecture for Experimental Research". In: *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*. Seattle, WA, USA, 2008.

[22] Teemu Rinta-aho, Mika Karlstedt, and Madhav P. Desai. "The Click2NetFPGA Toolchain". In: *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 77–88. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/rinta-aho.

[23] Anuj Kalia et al. "Raising the Bar for Using GPUs in Software Packet Processing". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 409–423. ISBN: 978-1-931971-218.

URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kalia.

[24] Sangjin Han et al. “PacketShader: A GPU-accelerated Software Router”. In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM ’10. New Delhi, India: ACM, 2010, pp. 195–206. ISBN: 978-1-4503-0201-2. DOI: 10.1145/1851182.1851207. URL: http://doi.acm.org/10.1145/1851182.1851207.

[25] *Google Project Ara*. http://www.projectara.com/.

[26] Gerard J. M. Smit et al. “Dynamic Reconfiguration in Mobile Systems”. In: *Proc. International Conference on Field-Programmable Logic and Applications (FPL)*. 2002.

[27] Mario Barbareschi, Antonino Mazzeo, and Antonino Vespoli. “Network Traffic Analysis Using Android on a Hybrid Computing Architecture”. In: *Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing - Volume 8286*. ICA3PP 2013. Vietri sul Mare, Italy: Springer-Verlag New York, Inc., 2013, pp. 141–148. ISBN: 978-3-319-03888-9. DOI: 10.1007/978-3-319-03889-6_16. URL: http://dx.doi.org/10.1007/978-3-319-03889-6_16.

[28] D.Koch, C. Beckhoff, and J Teich. “Recobus-builder a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas”. In: *Proc. Field Programmable Logic and Applications (FPL)*. 2008.

[29] M. Majer et al. “The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-Based Computer”. In: *VLSI Signal Processing Systems*. 2007.

[30] E. Horta, J. Lockwood, and D. Parlour. “Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration”. In: *Proceedings of the 39th conference on Design automation*. 2002.

[31] *Orbot*. https://guardianproject.info/apps/orbot. URL: https://guardianproject.info/apps/orbot.

[32] Roger Dingledine, Nick Mathewson, and Paul Syverson. “Tor: The Second-generation Onion Router”. In: *Proc. USENIX Security Symposium*. San Diego, CA, 2004.

[33] Yuvraj Agarwal et al. “Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage”. In: *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2009.

[34] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. “Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications”. In: *Proc. ACM SIGCOMM Conference on Internet Measurement Conference (IMC)*. Chicago, Illinois, USA, 2009.

[35] K. Amiri et al. “WARP, a Unified Wireless Network Testbed for Education and Research”. In: *Proceedings of IEEE MSE*. 2007.

[36] Stephen Neuendorffer and Chad Epifanio. “Generic partially reconfigured processor systems applied to software defined radio”. In: *Proc. of the Software Defined Radio Forum (SDR)*. 2007.

[37] *Universal Software Radio Peripheral (USRP) by Ettus Research*. `http://www.ettus.com/`.

[38] David G. Andersen et al. “Accountable Internet Protocol (AIP)”. In: *Proc. ACM SIGCOMM*. 2008.

[39] Van Jacobson et al. “Networking Named Content”. In: *Proc. Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. Rome, Italy, 2009.

[40] *40Gbit AES Encryption Using OpenCL and FPGAs*. `http://www.nallatech.com/40gbit-aes-encryption-using-opencl-and-fpgas`.

[41] *FPGA System Smokes Spark on Streaming Analytics*. `www.datanami.com/2015/03/10/fpga-system-smokes-spark-on-streaming-analytics/`.

[42] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Jose, CA, 2012.

[43] Gordon Brebner. “Circlets: Circuits As Applets”. In: *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. 1998.

[44] Steven A. Guccione and Delon Levi. “XBI: A java-based interface to FPGA hardware”. In: *Configurable Computing: Technology and Applications, Proc. SPIE 3526*. 1998, pp. 97–102.

[45] Eric Lechner and Steven A. Guccione. “The Java environment for reconfigurable computing”. In: *Proc. International Workshop on Field-Programmable Logic and Applications*. 1997.

[46] Gordon J. Brebner. “A virtual hardware operating system for the xilinx xc6200”. In: *Proc. International Workshop on Field-Programmable Logic (FPL)*. 1996.

[47] O. Diessel and G. Wigley. *Opportunities for operating systems research in reconfigurable computing*. Tech. rep. ACRC99018. Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, 1999.

[48] J-Y. Mignolet et al. “Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip”. In: *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*. 2003.

[49] Hayden Kwok-Hay So and Robert Brodersen. “A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH”. In: *ACM Trans. Embed. Comput. Syst.* 7.2 (2008).

[50] Edson L Horta, John W Lockwood, and Saint Louis. *PARBIT : A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs)*. Tech. rep. WUCS-01-13. Dept. Comput. Sci., Washington Univ., Saint Louis MO, 2001.

[51] S.Guccione, D. Levi, and P. Sundararajan. “JBits: Java-based interface for reconfigurable computing”. In: *Proc. Conf. on Military and Aerospace Application of Programmable Devices and Technology*. 1999.

[52] E. Keller. “JRoute: A run-time routing API for FPGA hardware”. In: *IPDPS Workshops, ser. Lecture Notes in Computer Science*. Vol. 1800. 2000.

[53] C. Patterson et al. “Slotless module-based reconfiguration of embedded FPGAs”. In: *ACM Trans. Embedd. Comput. Syst*. 2006.

[54] `http://programmablelogicinpractice.com/?p=87`.

[55] `http://www.opensource.apple.com/source/CommonCrypto/CommonCrypto-55010/Source/libtomcrypt/src/ciphers/ltc_aes/aes.c`.

[56] `http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf`.

[57] Alex Paek and Duncan Mackay. *Implementing Carrier Phase Recovery Loop Using Vivado HLS*. `http://www.xilinx.com/support/documentation/application_notes/XAPP1173-carrier-loop.pdf`. URL: `http://www.xilinx.com/support/documentation/application_notes/XAPP1173-carrier-loop.pdf`.

[58] Larry McVoy and Carl Staelin. “Lmbench: Portable Tools for Performance Analysis”. In: *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*. ATEC ’96. San Diego, CA: USENIX Association, 1996, pp. 23–23. URL: `http://dl.acm.org/citation.cfm?id=1268299.1268322`.

[59] *Tor Source Code Hacking Documentation*. `https://gitweb.torproject.org/tor.git/tree/doc/HACKING`. URL: `https://gitweb.torproject.org/tor.git/tree/doc/HACKING`.

[60] Chen Chang, John Wawrzynek, and Robert W. Brodersen. “BEE2: A High-End Reconfigurable Computing System”. In: *IEEE Des. Test* 22.2 (2005).

[61] Myron King, Jamey Hicks, and John Ankcorn. “Software-Driven Hardware Development”. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’15. Monterey, California, USA: ACM, 2015, pp. 13–22. ISBN: 978-1-4503-3315-3. DOI: 10.1145/2684746.2689064. URL: http://doi.acm.org/10.1145/2684746.2689064.

[62] Niranjan Soundararajan. “rSmart: The Reconfigurable (Real) Smartphone”. In: *Provocative Ideas session of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2013).

[63] Andrew Putnam et al. “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services”. In: *41st Annual International Symposium on Computer Architecture (ISCA)*. 2014. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=212001.

[64] Kalin Ovtcharov et al. *Accelerating Deep Convolutional Neural Networks Using Specialized Hardware*. 2015. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=240715.

[65] *Intel Altera Acquisition*. https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/.

[66] Prabhat K. Gupta. “Xeon+FPGA Platform for the Data Center”. In: *The Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)* (June 2015).

[67] *OpenCL*. https://www.khronos.org/opencl/.

[68] *Intel OpenCL SDK*. https://software.intel.com/en-us/intel-opencl.

[69] *Qualcomm Adreno GPU SDK*. https://developer.qualcomm.com/software/adreno-gpu-sdk/tools.

[70] *ARM Mali OpenCL SDK*. http://malideveloper.arm.com/resources/sdks/mali-opencl-sdk/.

[71] *GPGPU OpenCL API*. http://www.vivantecorp.com/index.php/en/technology/gpgpu.html.

[72] *PowerVR SDK*. https://community.imgtec.com/developers/powervr/.

[73] *Puzzlephone*. http://www.puzzlephone.com/.

[74] *Fairphone*. https://www.fairphone.com/.

[75] *LG G5*. http://www.lg.com/us/mobile-phones/g5.

[76] *Android On Zynq Getting Started Guide*. http://www.wiki.xilinx.com/Android+On+Zynq+Getting+Started+Guide.

[77] *Android 4.2.2 On Zynq Getting Started Guide*. http://www.wiki.xilinx.com/Android+4.2.2+On+Zynq+Getting+Started+Guide.

[78] *Zedroid - Android (5.0 and later) on Zedboard*. http://www.slideshare.net/noritsuna/zedroid-android-50-and-later-on-zedboard.

[79] *Intel Software Guard Extensions (SGX): A Researcher's Primer*. https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2015/january/intel-software-guard-extensions-sgx-a-researchers-primer/.

[80] *Intel Software Guard Extensions*. https://software.intel.com/en-us/sgx.

[81] Ferdinand Brasser et al. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, 2017. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

[82] Michael Schwarz et al. "Malware Guard Extension: Using SGX to Conceal Cache Attacks". In: *CoRR* abs/1702.08719 (2017). URL: http://arxiv.org/abs/1702.08719.

[83] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems". In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 640–656.

[84] Samuel Weiser and Mario Werner. "SGXIO: Generic Trusted I/O Path for Intel SGX". In: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. CODASPY '17. Scottsdale, Arizona, USA: ACM, 2017, pp. 261–268. ISBN: 978-1-4503-4523-1. DOI: 10.1145/3029806.3029822. URL: http://doi.acm.org/10.1145/3029806.3029822.

[85] Nico Weichbrodt et al. "AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves". In: *European Symposium on Research in Computer Security*. Springer. 2016, pp. 440–457.

[86] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. "Sanctum: Minimal RISC Extensions for Isolated Execution." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 564.

[87] Guy Gogniat et al. “Reconfigurable hardware for high-security/high-performance embedded systems: the SAFES perspective”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16.2 (2008), pp. 144–155.

[88] *Microsemi FPGA & SoC Security*. https://www.microsemi.com/products/fpga-soc/security.

[89] Andrew Baumann. “Hardware is the New Software”. In: *Proc. Workshop on Hot Topics in Operating Systems (HotOS)*. Whistler, BC, Canada, 2017, pp. 132–137.

[90] Intel Intel. “and IA-32 Architectures Software Developers Manual”. In: *Volume 3A: System Programming Guide, Part* 1.64 (64).

[91] Trusted Computing Group. *Trusted Platform Module Main Specification (TPM1.0)*. http://www.trustedcomputinggroup.org/resources/tpm_main_specification. 2011.

[92] Trusted Computing Group. *Trusted Platform Module Library Specification (TPM2.0)*. http://www.trustedcomputinggroup.org/resources/tpm_library_specification. 2013.

[93] *The Chromium Projecst: TPM Usage*. URL: http://www.chromium.org/developers/design-documents/tpm-usage.

[94] *Overview of BitLocker Device Encryption in Windows 10*. 2017. URL: https://docs.microsoft.com/en-us/windows/device-security/bitlocker/bitlocker-device-encryption-overview-windows-10.

[95] Sandeep Tamrakar et al. “Applications of Trusted Execution Environments (TEEs)”. In: (2017).

[96] Himanshu Raj et al. “fTPM: A Firmware-based TPM 2.0 Implementation”. In: *Microsoft Research* (2015).

[97] *ARM TrustZone*. https://www.arm.com/products/security-on-arm/trustzone.

[98] Jingfei Kong et al. “Deconstructing new cache designs for thwarting software cache-based side channel attacks”. In: *Proceedings of the 2nd ACM workshop on Computer security architectures*. ACM. 2008, pp. 25–34.

[99] Zhenghong Wang and Ruby B Lee. “New cache designs for thwarting software cache-based side channel attacks”. In: *ACM SIGARCH Computer Architecture News*. Vol. 35. 2. ACM. 2007, pp. 494–505.

[100] Fangfei Liu et al. “Catalyst: Defeating last-level cache side channel attacks in cloud computing”. In: *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 406–418.

[101] Siddhartha Chhabra et al. “SecureME: a hardware-software approach to full system security”. In: *Proceedings of the international conference on Supercomputing*. ACM. 2011, pp. 108–119.

[102] Ming-Wei Shih et al. “T-SGX: Eradicating controlled-channel attacks against enclave programs”. In: *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*. 2017.

[103] Ferdinand Brasser et al. “DR. SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization”. In: *arXiv preprint arXiv:1709.09917* (2017).

[104] Yangchun Fu et al. “SGX-LAPD: thwarting controlled side channel attacks via enclave verifiable page faults”. In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer. 2017, pp. 357–380.

[105] Daniel Gruss et al. “Strong and efficient cache side-channel protection using hardware transactional memory”. In: *USENIX Security Symposium*. 2017.

[106] Olga Ohrimenko et al. “Oblivious Multi-Party Machine Learning on Trusted Processors.” In: *USENIX Security Symposium*. 2016, pp. 619–636.

[107] Andrew Ferraiuolo et al. “Komodo: Using Verification to Disentangle Secure-enclave Hardware from Software”. In: *Proc. Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, 2017.

[108] Samuel Weiser and Mario Werner. “SGXIO: Generic Trusted I/O Path for Intel SGX”. In: *Proc. ACM on Conference on Data and Application Security and Privacy (CODASPY)*. Scottsdale, Arizona, USA, 2017.

[109] G Edward Suh et al. “AEGIS: architecture for tamper-evident and tamper-resistant processing”. In: *Proceedings of the 17th annual international conference on Supercomputing*. ACM. 2003, pp. 160–171.

[110] David Lie et al. “Specifying and verifying hardware for tamper-resistant software”. In: *Security and Privacy, 2003. Proceedings. 2003 Symposium on*. IEEE. 2003, pp. 166–177.

[111] David Champagne and Ruby B Lee. “Scalable architectural support for trusted software”. In: *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE. 2010, pp. 1–12.

[112] Jakub Szefer and Ruby B Lee. "Architectural support for hypervisor-secure virtualization". In: *ACM SIGPLAN Notices*. Vol. 47. 4. ACM. 2012, pp. 437–450.

[113] Dmitry Evtyushkin et al. "Iso-x: A flexible architecture for hardware-managed isolated execution". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2014, pp. 190–202.

[114] Thomas Wollinger, Jorge Guajardo, and Christof Paar. "Security on FPGAs: State-of-the-art Implementations and Attacks". In: *ACM Trans. Embed. Comput. Syst.* 3.3 (Aug. 2004), pp. 534–574. ISSN: 1539-9087.

[115] A. J. Elbirt et al. "An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9.4 (2001), pp. 545–557.

[116] Andreas Dandalis and Viktor K. Prasanna. "An Adaptive Cryptographic Engine for Internet Protocol Security Architectures". In: *ACM Trans. Des. Autom. Electron. Syst.* 9.3 (2004), pp. 333–353.

[117] Dino Oliva, Rainer Buchty, and Nevin Heintze. "AES and the Cryptonite Crypto Processor". In: *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES '03. San Jose, California, USA, 2003, pp. 198–209. ISBN: 1-58113-676-5.

[118] A. Hodjat and I. Verbauwhede. "High-throughput programmable cryptocoprocessor". In: *IEEE Micro* 24.3 (2004), pp. 34–45.

[119] *MicroBlaze Soft Procesor Core*. URL: https://www.xilinx.com/products/design-tools/microblaze.html.

[120] Daniel J Bernstein et al. "High-speed high-security signatures". In: *Journal of Cryptographic Engineering* (2012), pp. 1–13.

[121] Paul Selkirk and Joachim Strmbergson. 2015. URL: https://trac.cryptech.is/browser/core/rng/trng.

[122] *Introducing the Intel Software Guard Extensions Tutorial Series*. https://software.intel.com/en-us/articles/introducing-the-intel-software-guard-extensions-tutorial-series.

[123] Moxie Marlinspike. *Technology preview: Private contact discovery for Signal*. Signal, 2017. URL: https://signal.org/blog/private-contact-discovery/.

[124] *Exploiting the DRAM rowhammer bug to gain kernel privileges*. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[125] S.M. Trimberger and J.J. Moore. "FPGA Security: Motivations, Features, and Applications". In: *Proceedings of the IEEE* 102.8 (2014), pp. 1248–1265. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2331672.

[126] *Here Are the Most Popular Apps for Secure Messages*. http://fortune.com/2017/01/17/most-popular-secure-apps/.