Breaking the Trust Dependence on Third Party Processes for Reconfigurable Secure Hardware

Aimee Coughlin, Greg Cusack, Jack Wampler, Eric Keller, Eric Wustrow
University of Colorado Boulder

ABSTRACT

Modern CPU designs are beginning to incorporate secure hardware features, but leave developers with little control over both the set of features and when and whether updates are available. Reconfigurable logic (e.g., FPGAs) has been proposed as an alternative as it is both hardware, so can have similar capabilities at a reasonable performance degradation, and programmable, allowing customization of the secure hardware. This programmability, however, opens new attack vectors that allow an adversary to re-program the FPGA. Past attempts to solve this rely on a party maintaining a shared key with the FPGA, but these business processes to keep that key secret have been shown to be quite vulnerable.

In this paper, we propose a new mechanism which eliminates the trust dependence on third party processes. This new mechanism consists of a self-provisioning stage, where keys are generated internal to the FPGA and never exposed externally, coupled with a secure update mechanism which allows updates to be governed by a policy defined by the secure hardware application. To demonstrate, we fully implemented these mechanisms on a Xilinx Zynq UltraScale+ FPGA along with an example secure co-processor with remote attestation with a flexible root of trust (in contrast to Intel SGX which fixes the root of trust to be Intel). Our performance evaluation of two applications, a password manager and a contact matching application, illustrates using FPGAs is practical.

CCS CONCEPTS

• **Security and privacy → Key management**; **Tamper-proof and tamper-resistant designs**; • **Hardware → Reconfigurable logic and FPGAs**.

KEYWORDS

Secure Hardware; FPGA; Trusted Execution Environment; SGX

ACM Reference Format:
Aimee Coughlin, Greg Cusack, Jack Wampler, Eric Keller, Eric Wustrow. 2019. Breaking the Trust Dependence on Third Party Processes for Reconfigurable Secure Hardware. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293895

FPGA '19, February 24–26, 2019, Seaside, CA, USA

ACM ISBN 978-1-4503-6137-8/19/02…$15.00
https://doi.org/10.1145/3289602.3293895

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

Table 1: Comparing 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.

1 INTRODUCTION

Secure hardware provides many benefits for securing computing systems. It enables encrypting sensitive data where physical access to the device is required to decrypt it [7], authenticating data feed systems [41], scaling blockchain transactions [26], and has the promise to address many of the security challenges with cloud computing [15]. However, despite the potential benefits, we are stuck with a constrained ecosystem of secure hardware providers.

Due to the cost, time, and complexity of designing and manufacturing proceessor hardware [4, 5], the design choices and trade-offs are decided unilaterally by the small set of chip manufacturers. This results in scattered support of a wide range of features, and ultimately limited selection for users of secure hardware. Table 1 presents a summary of several secure hardware systems and the features they choose to support. Even in this modest set of features, there is no existing system that offers every feature, despite each system implementing features the other does not.

Furthermore, updates to secure hardware systems in response to discovered vulnerabilities [11, 13, 14, 17, 33, 36, 37, 40] 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.

In this paper, we seek to empower the individuals that ultimately use secure hardware to make decisions that are right for their needs, rather than the hardware manufacturers making choices for them.

Prior research has proposed that programmable hardware, such as field-programmable gate arrays (FPGAs), are suitable for implementing security functions [19–23, 28, 29, 32, 35]. FPGAs are programmable, providing flexibility to define the exact features that are needed, while allowing updates and retaining the performance

benefits of hardware [20, 23]. Importantly, FPGAs are no longer special purpose devices, but becoming pervasive in computing platforms such as cloud computing (*e.g.,* Amazon [1] and Microsoft data centers [10, 18, 31]), and in embedded systems for which secure hardware can provide great benefits, such as self-driving cars [2].

The programmable nature of FPGAs, however, raises a significant concern with regards to using them as a basis for realizing secure hardware – an attacker can read or modify the contents of the FPGA. This is in contrast to secure hardware systems built into silicon, which are "fixed", and cannot have their functionality changed after manufacture. Modern FPGAs include hardware that supports encrypted bitstreams [9, 39]. While an improvement, we argue that this doesn't completely solve the problem, but this only reduces the control of reprogrammability to a single party. This party is responsible for generating and maintaining the keys that protect access and functionality of the device. In other words, it depends on human / business processes, which, as history has shown with the frequent password and other data leaks [38] (including secure boot keys [11]), cannot be counted on.

In this paper, we introduce a novel mechanism to address this problem where we build on the capabilities provided by modern FPGAs and *put the device itself in control over the programmability*, thus removing the trust dependence on a third party's processes and providing developers with control over how the secure hardware is protected. This consists of two key aspects. The first is a self-provisioning mechanism where a device is initially brought up in a provisioning configuration, and then internally generates keys, and reprograms itself using these keys. In this way, the keys which control the configuration of the FPGA are only accessible internal to the device. The second is a policy driven update mechanism, where the hardware running in the FPGA is programmed with a policy which determines under what conditions to allow an update. In this way, we empower the secure hardware developer with the choice for how updates can occur (which could include a policy to block all updates). This allows the developer to choose (and commit to) how updates are (or aren't) performed on the device, allowing them to decide between a locked-down design similar to silicon-based secure hardware, or leaving systems flexible once deployed.

We demonstrate that this new mechanism is practical today with off-the-shelf FPGAs. Our implementation uses the Xilinx Zynq UltraScale+ MPSoC FPGA on the ZCU102 board. The application of this is broad, but as a single running example, we implement a secure coprocessor with an Intel SGX-like remote attestation feature. Unlike SGX's attestation, our remote attestation is designed to allow the device provisioner to choose who the root of trust is (rather than Intel's fixed root of trust being Intel), allowing for a wider range of trusted third parties to enable verified remote execution. We further use this running example to enable updates, which are motivated in this case to enable a response to newly discovered vulnerabilities, such as Spectre [25]. We provide an SDK to compile programs to execute in this secure co-processor environment. Unique to this FPGA environment, we can compile the developer's C code to either hardware using high-level synthesis, or to software to run on a soft processor (a CPU implemented using the FPGA logic). We built two applications on top of this customized secure co-processor – a password manager (similar to the example in the Intel SGX tutorial), and a contact matching application (emulating the SGX-enabled private contact discovery service operated by Signal [27]).

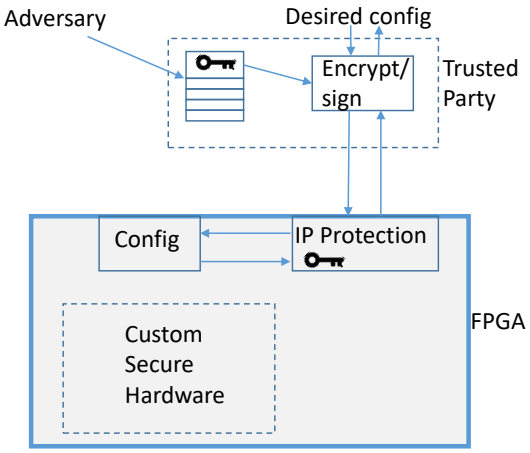

Figure 1: Custom secure hardware on an FPGA with IP protection. A designated party shares a cryptographic key with the FPGA which is used to ensure only FPGA configuration signed/encrypted with this key can re-program the FPGA. The designated party uses processes to protect the storage of the key, but an adversary can attack those processes and gain access to the shared key.

In the remainder of the paper we first discuss the past efforts of secure hardware on FPGAs (Section 2). We then provide an overview of the system architecture, threat model, and motivating example in Section 3. We describe the the architecture in Sections 4 and 5. We then describe the implementation of the self-provisioning and secure update mechanism (Section 6) and the secure co-processor with remote attestation (Section 7). We wrap up with evaluation (Section 8), and conclusions and future work (Section 9).

2 PAST ATTEMPTS (AND WHY PROCESS TRUST MATTERS)

In this paper we propose using FPGAs as a platform to build secure hardware. Here, we discuss past works, and identify the key unmet challenge in reaching this goal.

2.1 Security Functions on an FPGA

The idea of implementing security functions on an FPGA is not new. In fact, it has been proposed for decades. Research has been published on everything from network security applications (*e.g.,* firewalls [28] and intrusion detection [35]) to cryptographic algorithms [21]. More recently, and highly related to our motivating examples, the SAFES architecture demonstrated the use of FPGA components to provide security primitives and guarantee invariants in program execution [23], and Sanctum is a RISC ISA extension realized on an FPGA that mitigates software side-channels and protects DRAM access [20].

Although these examples demonstrate the ability to implement security functions on an FPGA, they do not address the somewhat obvious threat of an adversary who reprograms the FPGA, changing the device configuration and functionality. We argue that for many secure hardware applications, this is a particularly important threat

to address. For instance, if a device manufacturer wishes to offer remote attestation features (such as in Intel SGX) or hardware-protected keys for hardware security modules (HSMs), their design must protect against an adversary with physical (or remote) control over the device after its initial configuration.

By default FPGA's provide no protection to their configuration, allowing an adversary to read or reprogram whatever functionality is placed in it, allowing them to read out sensitive keys or change the device's behavior.

2.2 Security Functions with Bitstream Encryption

In response to this, FPGA manufacturers introduced bitstream protection technology, whether for intellectual property (IP) protection or specifically to support secure hardware [9, 39]. As illustrated in Figure 1, a third party programs a key into the FPGA and then maintains that key (external to the FPGA) so that it can be used to create an FPGA configuration that is encrypted and/or signed. In this way, knowledge of that key is needed to program the FPGA or read its configuration.

While an improvement, it fundamentally depends on a human-driven / business process to protect the key that is programmed into the FPGA. Unfortunately, this has proven to be a challenging problem and particularly fragile means for security. We have seen countless data leaks, including passwords [38] and even secure boot keys [11] (things that we *should* be able to assume won't be leaked). In addition, governments can compel key-holders to divulge their secrets in order to attack individuals, such as in the FBI vs. Apple [3], ultimately undermining end-user trust in the systems. In short, IP protections only serve to focus an adversary's efforts on the process, and once successful would still be able to read or modify any FPGA that was under the 'protection' of that party.

3 SYSTEM ARCHITECTURE

3.1 High-level Overview

We present our high-level design which eliminates the human / business processes from the trust chain. We do this by designing the FPGA to have control over its own reprogrammability, and allowing it to determine when (or if) to allow updates to itself. This design eliminates the need for a trusted party to maintain keys through a business processes, which we argue has historically been shown to be problematic.

The self-provisioning system is designed to allow the device to be initially provisioned once by a system manufacturer into a secure state, and thereafter prevent any future updates externally. To do this, we leverage existing secure hardware systems used for IP protection (*e.g.,* secure boot) that controls the boot process of the device. We configure the secure boot to only allow a single configuration to be loaded into the FPGA. This configuration effectively locks out external access, preventing an adversary with physical access from changing the hardware loaded into the FPGA. Once in this state, **not even the original manufacturer can directly change the configuration**. The private keys used to sign this configuration are generated on the FPGA during provisioning, and stored in a secure storage that is only accessible to the FPGA itself once booted. Because secure boot prevents loading arbitrary

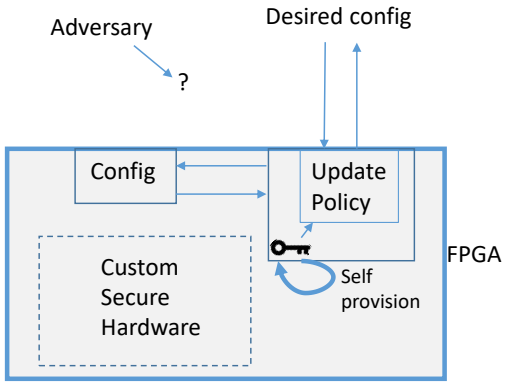

Figure 2: Secure Hardware on an FPGA with Self-Provisioning and Secure Updates. As the keys are only held within the FPGA, and updates are governed by hardware that implements an update policy, an adversary cannot gain access to the key or re-program the FPGA.

bitstreams into the FPGA, nothing except the FPGA itself has access to the secret keys needed to sign new bitstreams.

This self-provisioning process prevents any future updates from being applied from an external source, but still allows the device itself to authorize and apply updates. We note that a developer could decide to disallow updates entirely by programming a configuration that simply discarded its own key, and gain the benefits of silicon-based secure hardware. However, should the developer wish to leverage the reprogrammability of the FPGA, they can choose to do so. If they do, the FPGA is configured with a subsystem for authorizing and applying updates to itself. This subsystem can implement security policies that are more powerful than simply giving up a remote key to the manufacturer. For example, in addition to a signed update from the manufacturer, the subsystem could determine if it is currently in a certain unlocked or safe state, or could require the user to authorize an update explicitly before it signs the new hardware and reprogramms itself. This architecture allows a manufacturer to commit to a security policy, and force themselves (and would-be adversaries) to follow these.

3.2 Threat Model Overview

The adversary in our model is someone who desires to modify the secure hardware implemented in the FPGA or to read back state of the secure hardware implemented in the FPGA. Our work seeks to solve the problem of trusting an external party with maintaining keys that protect the FPGA configuration/state from this adversary. This requires a distinction between trust in operational processes and trust in functionality. In particular, we assume that the FPGA manufacturer is trustworthy at the time the device is created and provisioned, but that the manufacturer may become untrusthworty at a later time, either by being compromised, legally compelled, or having shifting business priorities. Thus, we assume that the original functionality of the FPGA as initially provisioned contains no backdoors or other malicious components, but that any long-term keys maintained by the manufacturer can be compromised.

We ignore the threat of implementation bugs in the secure hardware application, and side-channels on the FPGA that may inadvertently compromise the security of the system [24]. Though likely to exist, we stress two points: first, existing secure hardware also suffers implementation bugs and side channel attacks, and second, our architecture is better able to handle these problems by allowing comprehensive updates.

3.3 Motivating Example

As a motivating example of customized secure hardware, we will focus on a secure co-processor with remote attestation. While there are other applications that can be built using our design, secure co-processors are a powerful example that enables a wide range of security applications.

Intel Software Guard Extensions (SGX) [4, 5] is an extension introduced by Intel to their CPUs which provides a *Trusted Execution Environment* (TEE), allowing developers to write software that executes in a context isolated from the rest of the system, including the operating system. SGX also supports remote attestation of the software running in this TEE, but is designed to only allow Intel to verify remote attestations. Others that use SGX for remote attestation must trust Intel to verify that a remote system is running the code it claims to be running.

In our motivating example, say a company needs SGX-like capabilities, but wishes to use a different party (or even itself) as the trusted source which provides the proof and verification needed in the remote attestation process. This is not possible with Intel (or any existing systems today), so this company would use or design a secure co-processor targeted at an FPGA that provides a TEE with remote attestation. When combined with our self-provisioning system with updates, they can trust that an adversary will not be able to alter their design and, by extension, trust that their TEE will behave as they designed.

This company also wishes to be able to respond to vulnerabilities and deploy patches to their secure co-processor. This comes from experience, as there are numerous examples of vulnerabilities discovered in secure hardware after its release [11, 13, 14, 17, 33, 36, 37, 40]. With the ability to update, the company protects itself from being locked into a vulnerable system or needing to recall physical hardware. Updates, they determine, should be signed by them and should also be verified by their users through the use of a PIN provided in a separate (assumed secure) channel to the user.

In Section 7 we will discuss our implementation of this specific co-processor system.

4 SELF-PROVISIONING

The goal of this work is to ensure that we can program an FPGA with a configuration implementing some custom secure hardware and trust that a malicious party cannot modify it. On the surface, secure boot would appear suitable for this. A secure boot system operates by verifying a signature over a booted configuration against a public key programmed into the system's configuration, such as a secure storage device. The trusted developer has the corresponding secret key and is theoretically the only party that can generate a correctly signed configuration. However, if this secret key is leaked to another party, then this party can put any configuration into the device.

Our solution still makes use of the IP protection hardware used by prior work [30, 39], but changes how the secure boot keys are managed. The problems with the use of secure boot are not related to how the hardware is implemented – the IP protection hardware was never compromised. It is the business processes that are used to protect the keys that we eliminate. Our self-provisioning system achieves this by generating the secure boot key pair on the device and storing the secret key in the device's storage. The system uses this key to sign a single initial configuration, which then becomes the only configuration that can exist in the FPGA.

The self-provisioning system is simply a trusted piece of software that is run on the device itself to generate keys which will be stored on the device and never exposed.

First, the FPGA is empty with no secure boot set up. The *self-provisioner* configuration is loaded and executes a series of steps, as summarized below:

(1) Generate a keypair for the secure boot system.
(2) Sign the *initial* FPGA configuration with the generated secret key.
(3) Store the secret key in secure storage.
(4) Program the public key to the secure boot system on the device.

At this point, the FPGA's secure boot has been set up and the keys are stored in secure storage on the device. Only the single configuration, determined at provisioning time is allowed to be loaded as it is the only one which has been signed by the secure boot keys. A power cycle of the device will then cause this *initial* configuration to be loaded onto the FPGA. In order for a different configuration to be loaded, it must be signed by the secret that only exists on the device and must be authorized by the security policy of the update mechanism (discussed in the next subsection) of this initial configuration.

The *initial* configuration could be the desired secure hardware application itself (*e.g.,* the secure co-processor with remote attestation), if known at provisioning time. If unknown, or if more flexibility is desired, an option would be to load an initial configuration that does not have any secure hardware application, but can have an update policy that suits the protection desired until loaded with the initial application (*e.g.,* a one-time use key). The update system would then be used to load the actual secure hardware application onto the FPGA. Note that this will result in overwriting the update system's policy with that of the secure hardware application's policy.

5 POLICY CONTROLLED SECURE UPDATES

The secure update system provides the second component of our platform that allows for applications to make use of the FPGA's reprogrammability. As described in the previous section, once self-provisioning is complete, only a single configuration can exist in the FPGA. However, since the generated secret is accessible to the FPGA, the FPGA can authorize a new configuration. Therefore, to allow for updates, a subsystem needs to be implemented by developers that will implement a security policy. This subsystem will receive updates and will verify that they conform to the selected security policy before using the secret key to authorize an update.

The update subsystem will enforce a security policy, but this policy must be selected and implemented by the developer of the application. Examples of security policies are:

- Update signed by a trusted developer.
- Correct PIN input by user at update time.
- User PIN and trusted signature required.
- No updates allowed.

This list of policies is not exhaustive, but is representative of potential policies. What this enables is choice for the secure hardware developer. They could trust their own processes (to safeguard keys), or, better yet, safeguard against leaks by utilizing a policy which requires signing *and* a PIN, and perhaps extend the policy to allow a new key for signing updates to be regenerated through some local action.

To support this, we require the developer to implement the enforcement of the chosen policy as part their application. This is because these implementations depend heavily on the capabilities of the device and developers will have their own requirements, such as signature algorithms or input devices, that cannot be prescribed for all use cases. We give an example implementation that is not portable outside of our device used for implementation in the next section, but can be used as an example to build other update systems off of, even when implementing a different security policy.

In general, the secure update system is responsible for performing two tasks, irrespective of the implementation or chosen policy. The first task is to receive updates and enforce that these updates adhere to the security policy before allowing them to be authorized (such as verifying a signature or user PIN). The second task is to use the device-only secret key to sign updates that pass verification and program the signed update to the device. Therefore, an update subsystem must perform these steps:

(1) Receive an update.
(2) Verify that the update conforms to the update security policy.
(3) Use the secret key to sign the update.
(4) Overwrite the existing FPGA configuration such that the update will execute in future power cycles of the device.

As the update system is implemented as part of the initial configuration of the FPGA that is authorized by the self-provisioning system, there is no other way to change the configuration. Therefore, the configuration is secure from being overwritten except by another update that conforms to the chosen policy. This requires that the developer implements the update policy correctly, as there are several attacks, such as man-in-the-middle, downgrade and rollback attacks, that can compromise a security policy that performs only simple authentication. Therefore, update best-practices should be followed, such as the use of sequence numbers and signatures, when implementing a security policy. This is further discussed in the next section, where we discuss which attacks that the update policy we implemented defends against and which it is still vulnerable to.

6 IMPLEMENTATION

To demonstrate our platform, we implemented a self-provisioning system and an example application that includes an update subsystem. Our example application is a secure coprocessor that offers similar features to SGX, and is described further in the next section. In this section, we present how we implemented the self-provisioning system and the update subsystem, which any implementation of our platform will need to provide. We also describe the implementation of a secure storage capability in our device, as both the self-provisioning system and the update system require a secure storage system. We implemented our demonstration application using the Xilinx ZCU102 Evaluation Kit. This system combines a quad-core ARM CPU and a Xilinx FPGA and includes all of the needed IP protection hardware that is required for our platform.

6.1 Self-Provisioning

In an ideal system, the FPGA would have direct internal control over the IP protection hardware, with all other peripherals restricted from accessing these systems. However, we were limited by the device we used for our implementation, in that the FPGA does not have direct access to most peripherals in the device's interconnect design. Instead, the coupled ARM CPU is the master of the system, meaning that our provisioning system needed to be run as a software program rather than as a system in the FPGA. This imposes some increased risk of exposure of generated keys, as the ARM system memory is more accessible than the FPGA, but since the self-provisioning system is expected to execute in a trusted facility, this increased risk can be mitigated.

The self-provisioning system that we implemented performs the tasks outlined in the previous section. The provisioner (*e.g.,* the device manufacturer or distributor) will load the self-provisioner onto the device's persistent storage (in our case, an SD card) along with the initial FPGA configuration to be signed. We, acting as the provisioner, have generated the self-provisioning operating system using Xilinx's proprietary tools such that when the device is powered on, the provisioner is executed.

Once booted, the provisioner loads a simple Ubuntu filesystem that executes a single script. This script generates an RSA-4096 keypair for the secure boot system (the ZCU102 secure boot hardware uses 4096-bit RSA keys) and stores it securely. As the only persistent storage available on our device is the SD card, we also leverage additional IP protection hardware that is used for FPGA encryption. This hardware utilizes a small amount of secure storage (battery-backed RAM (BBRAM)) that cannot be read once it is programmed. The self-provisioning system generates an encryption key, programs the encryption key to the BBRAM, and uses the encryption key to encrypt the generated secure boot keypair. On each future boot, the encryption hardware can decrypt the secret key if needed without it being decryptable outside of the device.

Once the keypair has been generated and the secure storage initialized with the encryption hardware, the self-provisioner uses the keypair and Xilinx's tools to generate a signed boot image containing the initial FPGA configuration that is in the proprietary format used by our device. The output file is then placed onto the SD card so that it will be loaded on the next power cycle of the device. Finally, the self-provisioner will program the generated public key into the IP protection secure boot system of the device, locking the device to only being able to run the boot image that was generated, which contains the initial FPGA configuration.

At this point, the self-provisioner is finished and reboots the device. On the next boot, the signed FPGA configuration will be

running and will be the only hardware that can be loaded into the FPGA, as the secure boot system will not let any other configurations that are not signed by the key into the FPGA, and no other such configurations can exist, since the secure boot key only exists on the device itself.

6.2 Update System

As required by our platform's architecture, the self-provisioning system locks down our device so that only a single FPGA configuration can exist in the FPGA. To support updates, our platform requires that developers include an update subsystem that will implement a security policy, but we require that the developers provide their own implementations. This is because developers need to make application-specific and device-specific decisions about how to implement the system. In this section, we describe the implementation we used for our application that demonstrates what these application-specific and device-specific can be.

The update system that we provided implements the required functionality of our platform. We selected a security policy that requires a trusted signature over the update and the input of a user's PIN before the update will be accepted. The verification of the security policy is performed by the FPGA, but because the FPGA does not have direct access to the SD card on our device, and because the boot image format that the update must be converted to is also proprietary, the actual generation of the boot image cannot be done in the FPGA. Instead, when the FPGA authorizes an update, the device will reboot into a simple update operating system that is similar to the self-provisioning system previously described. This means that our update operating system is implemented partially in the authorized FPGA configuration, but also in the update operating system and a trusted bootloader.

When an update is authorized, the update subsystem will store a flag into the secure storage that is only accessible to the FPGA. Upon reboot, the bootloader will check for the existence of this flag and boot into a different operating system. This update system in the FPGA will then release the private key to the operating system after the trusted bootloader indicates that it has booted. The update operating system's only task is to use the secret key and Xilinx's tools to generate a compatible boot image that contains the updated FPGA configuration. Once it has generated this boot image and placed it into persistent storage, the operating system will reboot the device into normal operation.

As can be seen, our device has several limitations that require special implementation considerations, specifically the fact that the FPGA does not have direct access to most system peripherals. In addition, for the enforcement of our security policy, we require user PINs to be six digits in length and we require all updates to be signed using the ED25519 signature algorithm. Other update systems may choose to use different requirements. We also make use of the MicroBlaze soft CPU to implement the update system, whereas other implementations may choose to use other methods, such as pure Verilog or a different CPU. Because of these considerations, we do not provide a single implementation, as any implementation depends upon the capabilities of the device, the requirements of the application, and the exact update security policy that is chosen.

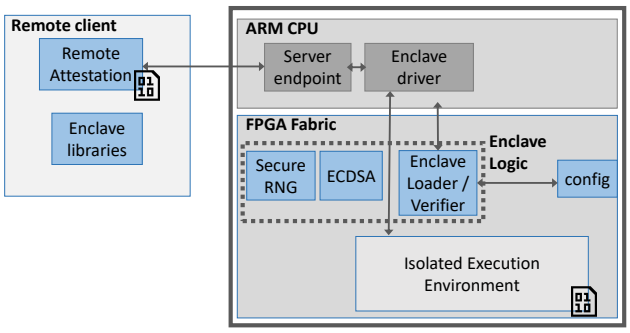

Figure 3: Secure Coprocessor and Remote Attestation Design. Here we run the FPGA as a coprocessor and are able to enforce isolation and perform remote attestation. A remote attestation client uploads a program to an untrusted server. The program is launched in a Isolated Execution Environment in the FPGA by enclave logic, which also signs the program code and performs a key exchange. The driver communicates with the program in the enclave over a shared buffer and relays data to the client.

6.3 Secure Storage

As mentioned in the previous two sections, the self-provisioning system and the update system both need to store secrets that are only accessible to the FPGA. However, our device does not provide such a capability directly, nor does it allow for the FPGA to directly write to the SD card. To solve this problem, we leverage the built-in encryption hardware, as mentioned previously, in the form of an AES accelerator that is backed by a secure encryption key storage in BBRAM. The self-provisioning system initializes this accelerator with a random key that never is stored except in the BBRAM and uses the accelerator to encrypt data. Using the accelerator, we can achieve a secure storage that prevents data from decrypted outside of the device.

However, the FPGA cannot directly pass data to the AES accelerator. Instead, we require that a proxy be run in the CPU of our device that passes data between the FPGA and the AES accelerator, and stores the encrypted data onto the SD card. To further protect the data, we have also implemented a corresponding subsystem in our application that interacts with this agent, which encrypts any arbitrary data generated by our application using an FPGA-only key that is stored in a dedicated eFuse array only accessible to the FPGA. This ensures that when passing data to the CPU agent after boot that no cleartext data is available in the CPU's memory.

7 A CUSTOMIZED SECURE COPROCESSOR WITH REMOTE ATTESTATION

In Section 3 we described a motivating example where a company wishes to have a secure co-processor with remote attestation where the root of trust is flexible (*i.e.,* not the manufacturer, as in SGX). In this section we elaborate on the hardware design, the software development kit to develop software applications, and two example software applications (password manager and contact matching) that were built with our software development kit.

7.1 Hardware Design

7.1.1 Isolated Execution Environment. The code that can be provided to the secure co-processor to run in an isolated manner is in the form of a partial configuration bitstream. There are two options we support for the internal architecture of this hardware. The surrounding logic is identical in both cases, but it is the contents of the configuration bitstream which differ.

Option 1: Software Enclave.

To provide a software environment for software isolation and remote attestation, we implemented a MicroBlaze [8] 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. Developers provide their code to the SDK, which will then generate the needed logic to execute this code in a MicroBlaze CPU.

Option 2: Hardware Enclave.

Alternatively, developers can directly provide hardware, so long as it is able to perform the interaction with the untrusted software. This does not imply the developer has to develop hardware. They can develop logic directly for the FPGA in any manner that they choose, including by synthesizing the developer's software (C code) into a compatible bitstream using high-level synthesis, as is described in Section 7.2.

The developer can make the decision between having their enclave's code (provided as C code) synthesized to hardware or executed on a soft CPU based on the complexity of the application – more complex applications are more difficult to synthesize to hardware, but an application synthesized to hardware will have better performance. The SDK will generate a resulting partial bitstream based on the developer's choice and the synthesis results that either includes the application directly implemented as FPGA logic, or a soft CPU in the FPGA logic that executes their application's code. The SDK also generates an untrusted program (*i.e.,* the "Enclave driver") that runs on the device's (untrusted) CPU to interact with the enclave program via a memory buffer in the FPGA.

7.1.2 Enclave Code Loader. In order to securely program this co-processor, we utilize custom logic that ensures that when any trusted code (*i.e.,* a trusted "enclave" program, similar to SGX) is loaded, a hash of this program is taken and a signature verification are performed. As illustrated in Figure 3, the code of the application is provided to the logic in the form of a partial bitstream, which specifies a configuration which will reprogram only part of the FPGA. The enclave logic will use the internal configuration access port (ICAP) to program the partial bitstream (the enclave program) into the area of the FPGA reserved for executing the secure enclave, leaving the rest of the FPGA (*e.g.,* enclave logic) untouched.

In addition, the enclave logic reads an ECDSA private key from the secure storage, and uses it to sign the hash of the bitstream and a message from the enclave during the remote attestation process. As shown in Figure 3, 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.

7.1.3 Remote Attestation. The attestation protocol implemented by our secure hardware and companion software is shown in Figure 4. In this protocol, a remote verifier uploads a program (in the form of a partial bitsream) signed by its Ed25519 private key (SK_v) [16]). The program will be launched by the enclave logic, and the verifier will be notified upon completion. The verifier will then request an attestation by uploading its signed public key (PK_v). The enclave logic then 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 using $PK_{enclave}$ 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.

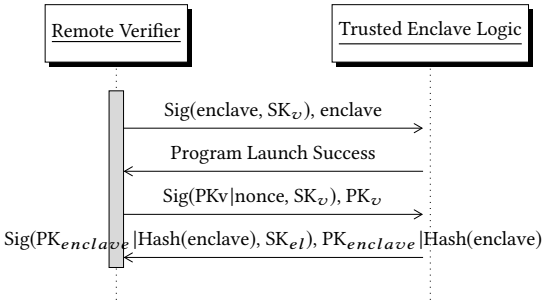

Figure 4: Remote Attestation Sequence: In the remote attestation protocol, the remote verifier uploads a program (enclave) signed by its private key (SK_v). The enclave launches the program and notifies the verifier, which then requests an attestation by sending its signed public key (PK_v). The enclave 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}).

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 [34]. 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 it. The resulting digest is used to seed a ChaCha stream cipher's key and IV which is used as a PRNG to provide random numbers to the enclave logic to securely generate keypairs.

7.2 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 5 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 enclave (enclave.c),

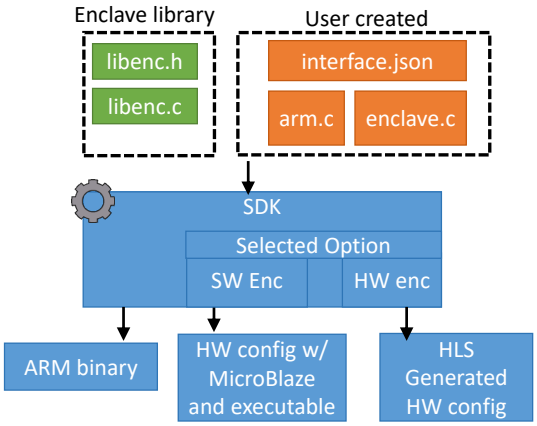

Figure 5: SDK Development Flow

and a 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 (libenc.h, libenc.c) that provides functions to launch an enclave, which is done by interacting with the enclave logic.

The developer provides their code to the SDK. For a software enclave, the SDK will output a partial configuration bitstream (which was pre-built) that contains a MicroBlaze [8] soft CPU (*i.e.,* a processor implemented in the FPGA logic). The SDK will cross-compile the enclave code and add the memory to the configuration bitstream. For a hardware enclave, the SDK will utilize the Vivado [12] high-level sythesis tool, which generates Verilog from C code. Then it will synthesize that design and generate a partial configuration bitstream.

In both cases, the SDK will use the API interface definition to generate communication code between the enclave and the ARM CPU using the dedicated shared buffer. Also, in both cases, the SDK will cross-compile the application code for the ARM instruction set. The (untrusted) ARM binary's will load the trusted code into the enclave using the enclave library.

7.3 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 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.

To use the manager, a user provides their master password to a client program which interacts with the enclave. The user then 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 achieved 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 [6].

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 8.

7.4 Contact Matching Application

As a second example to show how our isolated environment can execute code that has been synthesized into FPGA logic using high-level synthesis, we have developed a second application. This application emulates the SGX-enabled contact discovery service operated by Signal [27], except implemented using C++ and synthesized into hardware using our SDK. This application's purpose is to receive an encrypted list of contacts (*i.e.,* phone numbers) from a user and determine the intersection of this with a database of all registered users of the service. The solution used by Signal is designed to prevent the operators of the service from learning the contacts in the uploaded list while still allowing for users to determine the intersection with the total database. By executing in an SGX enclave, Signal is able to conceal which contacts are found to match, and return an encrypted result to the user. Our contact matching application provides similar functionality, but executes its code in FPGA logic that has been synthesized using our SDK. We present the performance of this application in Section 8.

8 EVALUATION

As an example secure hardware application, we built a secure co-processor with remote attestations. Here, we we evaluate the performance of example applications for this secure co-processor along with associated metrics about how long it takes to load and perform a remote attestation. For all of our applications we continue to use the ZCU102 Evaluation Kit running Ubuntu 15.10.

8.1 Software 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.

Software 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.

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 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.

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 7.3, for passwords of up to 100 characters in length. As seen in Figure 7, both with and without running in an enclave results in an average 202ms latency (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).

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. 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).

8.2 Hardware Enclave Performance

To show that our SDK can also achieve acceptable performance for large scale processing, particularly through high-level synthesis (compiling C code directly to hardware), we developed a second application that performs a similar service as the contact discovery service operated by Signal. As discussed previously, the purpose of our application is to receive a list of phone numbers from a user and determine the intersection with a larger database, and then return the result to the user. We compared the performance of this application to a software-only implementation that used the same contact list and database. As shown in Figure 6, the synthesized hardware version achieves a throughout of up to 3x compared to the software solution. We used contact list sizes of 128 contacts, represented as SHA512 hashses, and database sizes ranging from 800 contacts to 819,200 contacts, also represented as SHA512 hashes.

8.3 Enclave Logic Microbenchmarks

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.

Remote Attestation Performance. To measure the end-to-end performance of performing a remote attestation, we implemented a

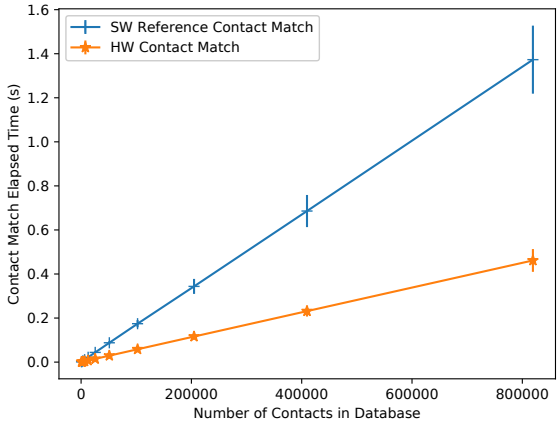

Figure 6: Contact Matcher Performance Performance of matching a contact list against a larger database in a software-only implementation and an HLS-synthesized version. The hardware version achieves an average of approximately 3x compared to the software version.

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 using the shared secret negotiated by the remote attestation protocol, and the other provided to the enclave by the local host, similar the contact discovery feature used by Signal [27]. 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.

9 CONCLUSIONS AND FUTURE WORK

In this paper we introduced a new mechanism which allows FPGAs to be used to implement customized secure hardware without depending on human / business processes to maintain the secrecy of keys used to protect the FPGAs configuration process. We introduced the concept of self-provisioning and a secure update process which allows for policies which govern whether an update is allowed or not. As a proof of concept, we implemented the framework on the Xilinx Zynq Ultrascale+ FPGA and built a secure co-processor with remote attestation that has a flexible root of trust. Going forward, a key direction we intend to pursue is to further strengthen the threat model and seek to more completely decouple the underlying mechanisms from the secure hardware applications – that is, modify the design such that the framework can provide run-time support for loading secure hardware applications, rather than the current boot-time support.

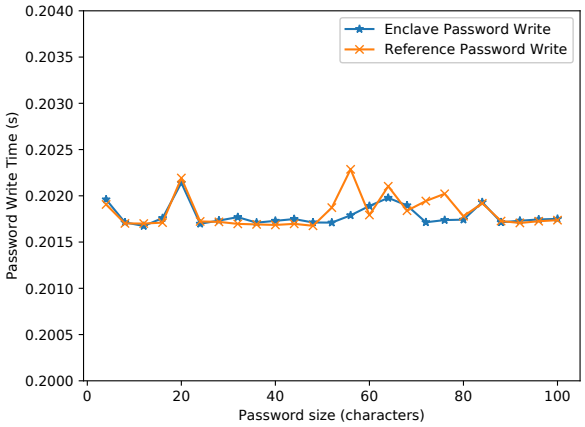

Figure 7: Password Manager Write Performance Time spent adding passwords to the password manager when protected by an enclave and when using a reference implementation running completely on the ARM CPU without an enclave.

Acknowledgements. We thank the anonymous reviewers for their input on this paper. This research was supported in part by the National Science Foundation under grants 1406192 (SaTC) and 1700527 (SDI-CSCS).

REFERENCES

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

[2] Ces: Intel goes for self-driving cars. https://www.electronicsweekly.com/news/design/ces-intel-goes-self-driving-cars-2017-01/.

[3] FBI Apple encryption dispute. https://en.wikipedia.org/wiki/FBI\T1\textendashApple_encryption_dispute.

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

[5] 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/.

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

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

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

[9] Microsemi: Security. https://www.microsemi.com/product-directory/fpga-soc/1738-security.

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

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

[12] Vivado user guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf.

[13] CVE-2016-3287. Available from MITRE, CVE-ID CVE-2016-3287, July 2016.

[14] CVE-2016-3320. Available from MITRE, CVE-ID CVE-2016-3320, Aug. 2016.

[15] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.*, 33(3), Aug 2015.

[16] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, pages 1–13, 2012.

[17] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[18] A. M. Caulfield et al. A cloud-scale acceleration architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016.

[19] P. Chodowiec and K. Gaj. Implementation of the twofish cipher using FPGA devices. Technical report, Electrical and Computer Engineering, George Mason University, 1999.

[20] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal risc extensions for isolated execution. *IACR Cryptology ePrint Archive*, 2015:564, 2015.

[21] A. Dandalis, V. K. Prasanna, and J. D. Rolim. A Comparative Study of Performance of AES Final Candidates Using FPGAs. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2000.

[22] A. J. Elbirt and C. Paar. An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher. In *Proc ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2000.

[23] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin. Reconfigurable hardware for high-security/high-performance embedded systems: the safes perspective. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(2):144–155, 2008.

[24] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine. Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems. In *IEEE Security and Privacy*, 2007.

[25] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

[26] J. Lind, I. Eyal, F. Kelbert, O. Naor, P. R. Pietzuch, and E. G. Sirer. Teechain: Scalable blockchain payments using trusted execution environments. *CoRR*, abs/1707.05454, 2017.

[27] M. Marlinspike. Technology preview: Private contact discovery for signal. https://signal.org/blog/private-contact-discovery/, 2017.

[28] J. T. McHenry, P. W. Dowd, F. A. Pellegrino, T. M. Carrozzi, and W. B. Cocks. An FPGA-based coprocessor for ATM firewalls. In *Proc IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM))*, 1997.

[29] S. McMillan and C. Patterson. Jbits implementations of the advanced encryption standard (rijndael). In *International Conference on Field Programmable Logic and Applications*, pages 162–171. Springer, 2001.

[30] E. Peterson. XAPP 1323: Developing Tamper-Resistant Designs with Zynq UltraScale+ Devices. https://www.xilinx.com/support/documentation/application_notes/xapp1323-zynq-usp-tamper-resistant-designs.pdf, Aug 2018.

[31] A. Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proc. Annual International Symposium on Computer Architecuture (ISCA)*, 2014.

[32] M. Riaz and H. M. Heys. The fpga implementation of the rc6 and cast-256 encryption algorithms. In *Electrical and Computer Engineering, 1999 IEEE Canadian Conference on*, volume 1, pages 367–372. IEEE, 1999.

[33] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using SGX to conceal cache attacks. *CoRR*, abs/1702.08719, 2017.

[34] P. Selkirk and J. Strömbergson. https://trac.cryptech.is/browser/core/rng/trng.

[35] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system. In *Field Programmable Logic and Application*, 2003.

[36] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security*, pages 440–457. Springer, 2016.

[37] S. Weiser and M. 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, pages 261–268, New York, NY, USA, 2017. ACM.

[38] Wikipedia. List of data breaches. https://en.wikipedia.org/wiki/List_of_data_breaches.

[39] K. Wilkinson. XAPP 1267: Using Encryption and Authentication to Secure an UltraScale/UltraScale+ FPGA Bitstream. https://www.xilinx.com/support/documentation/application_notes/xapp1267-encryp-efuse-program.pdf, Aug 2018.

[40] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

[41] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.