PULL THE RUG FROM UNDER: MALICIOUS RECONFIGURATION OF EXECUTING PROGRAM IN FPGA AND ITS DEFENSE

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Michael Glapa

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Felix Lin, Co-Chair

School of Electrical and Computer Engineering

Dr. Saurabh Bagchi, Co-Chair

School of Electrical and Computer Engineering

Dr. Jan Allebach

School of Electrical and Computer Engineering

Approved by:

Dr. Pedro Irazoqui

Head of the School Graduate Program

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Glapa, Michael M.S., Purdue University, May 2019. Pull the Rug from Under: Malicious Reconfiguration of Executing Program in FPGA and its Defense. Major Professors: Felix Lin, Saurabh Bagchi.

The Field Programmable Gate Array (FPGA) has been used for decades in embedded applications where custom hardware is not practical or feasible. However, thanks to increases in size and compute capabilities, the FPGA has become more attractive as an option to supplement a general-purpose Central Processing Unit (CPU) for accelerating complex computations used for encryption, machine learning, and many other applications. Although FPGAs have already appeared in embedded Systems-on-Chip (SoC) and cloud environments, the reconfigurable nature of FPGAs creates security vulnerabilities not found in more traditional accelerators like Graphics Processing Units (GPU). In this paper, we describe a vulnerability in an Altera Cyclone V SoC and demonstrate an attack that exploits this vulnerability. We propose a hardware modification that would provide a defense against this attack, and we implement a Linux kernel module to demonstrate a proof-of-concept for this hardware solution.

1. INTRODUCTION

The reconfigurable nature of FPGAs yields distinct advantages over more traditional compute devices for certain applications. Since the logic within the FPGA can be optimized for a wide range of tasks, they can achieve lower latency, higher performance, and higher power efficiency than other accelerators. These advantages make FPGAs an attractive solution when additional compute power is required in environments ranging from datacenters to mobile devices. An FPGA accelerator can be used to supplement a general-purpose CPU in the form of a PCIe card or as an additional on-chip module in an SoC. These devices allow applications running within an operating system to offload complex computations to the device to be performed by optimized hardware. In this way, FPGAs can be used similarly to GPU accelerators. Although FPGAs have already begun to appear in such environments, existing security features fail to eliminate vulnerabilities which are unique to FPGA accelerators.

Like any other accelerator, the ability of an FPGA to perform a given algorithm depends on the logic implemented by the hardware. However, while the logic in a CPU, GPU, or other ASIC cannot be changed, an FPGA can be reconfigured any number of times to modify the functionality of the chip on a hardware level. Though this feature can be exploited to achieve higher performance and efficiency through heavy optimization, it can also be exploited by an attacker. If an FPGA is being used by an application to perform some computation, an attacker can reconfigure the FPGA to potentially crash the application, leak data, or alter the algorithm performed by the application.

There are several flaws inherent to the design of an FPGA which enable this attack. Firstly, an FPGA must be configured frequently. The Static Random-Access Memory (SRAM) registers that store the configuration on-chip are volitile, and so a legitimate user must reconfigure the device if it loses power. In addition, a user may

wish to reconfigure the FPGA often in order to optimize it for different tasks. To allow this, an FPGA accelerator can be reconfigured by software within an operating system or through direct physical access to the device. A user with the ability to reconfigure the FPGA through either of these methods would be able to attack an application running on the FPGA. The second flaw which enables this attack is the lack of ability to read back the current configuration on an FPGA. This ability is deliberately disabled by FPGA vendors in order to prevent IP theft. But because the configuration cannot be verified, it is impossible for a user to determine if the configuration of the FPGA has been tampered with.

A successful attack on an FPGA accelerator could cause serious problems for an application. An attacker can easily crash the application by simply reconfiguring the FPGA with a different bitstream, however a more sophisticated attack can allow the application to keep running but with altered functionality. The possible outcomes for such an attack are nearly infinite since the attacker can essentially rewrite the application to perform any task the device is capable of performing. The attack could potentially leak unencrypted data from an encryption application or alter the algorithm in a mission-critical application like a self-driving car.

In this paper, we propose a potential hardware modification which could defend against this attack, and we emulate this hardware modification using a custom kernel module. To prevent this attack, the defensive measures must still allow an authorized user to configure the FPGA as is required for typical operation of the device, and it must also disallow readback in order to prevent IP theft. With these limitations in mind, we propose the addition of an additional hardware device located on-chip which can store a cryptographic hash of the FPGA's configuration. By checking this hash value, a user or application can verify the configuration without obtaining information about the IP that could allow reverse engineering, cloning, etc.

To demonstrate a proof-of-concept for this hardware modification, we implemented a kernel module which functions similarly to the theoretical hardware security feature. The kernel module acts as a middleman between the operating system and the

FPGA. To configure the FPGA, an application sends the the bitstream to the kernel module which computes a hash of the bitstream before configuring the FPGA. The kernel module stores this hash value in a file which can be quickly checked by an application. By disabling JTAG configuration and restricting access to the FPGA hardware through Linux user groups, we can ensure that our custom kernel module is the only method of configuration. As such, the hash value will always accurately reflect the FPGA's current configuration. By polling this value, an application can detect an attack and react appropriately.

2. RELATED WORK

Although modern FPGAs contain a sophisticated set of security features [4], there is little defense against attacks on FPGA accelerators such as those found in embedded SoCs or cloud environments. Researchers have demonstrated a side-channel attack on FPGAs in a cloud environment [5], as well as an attack on an SoC containing an FPGA [6]. In addition, due to the prevalence of GPU accelerators in cloud platforms, researchers have investigated vulnerabilities pertaining to data privacy in a shared environment [7] [8]. Since FPGAs in the cloud generally take the form of a PCIe card similar to a GPU, these attacks on GPUs may be possible on FPGA accelerators as well.

3. BACKGROUND

In this section, we provide background information about FPGAs, describe existing security features for preventing attacks on FPGAs, and explain the use of FPGAs for accelerating computation.

3.1 FPGA Primer

An FPGA is an integrated circuit which consists of an array of logic blocks that can be configured to perform any task that an ASIC or general purpose CPU could perform, provided it has sufficient resources. Although there have historically been many types of FPGAs, the most common type uses SRAM to control the internal connections between the logic blocks. By setting specific values in the SRAM registers, a programmer can essentially "re-wire" the the blocks within the FPGA to optimize the performance of a specific task. Since SRAM FPGAs consist of only transistors and wires, they benefit from new process nodes much quicker than other types of FPGAs, yielding higher performance and lower power consumption. In addition, while some FPGA technologies utilize fuses that allow the FPGA to be configured only once, SRAM FPGAs can be configured an unlimited number of times. However, because SRAM is volatile, the FPGA loses its configuration when powered off, requiring it to be reconfigured when power is restored. Because of this, a product which utilizes an SRAM FPGA must incorporate a second device to configure it after the power is cycled. This additional complexity is the source of many FPGA attacks, since it indroduces the ability to steal or tamper with the configuration data. Creating a hardware design for an FPGA is expensive and time consuming, and the resulting design may include trade secrets or classified technology. To prevent attackers from

extracting information about the hardware design, modern FPGAs contain a complex set of security features.

3.2 FPGA Security Features

In order to configure an FPGA, the hardware design must be translated to a format which the FPGA can use to implement the logic. This configuration data is called the bitstream. Since the configuration of an SRAM FPGA is volatile, the bitstream must be stored on another device and sent to the FPGA once it is powered up. Because of this necessary data transfer, there are many attacks that can be attempted on the system. Most attacks on FPGAs attempt to either extract the hardware design or tamper with the configuration of the FPGA. Modern FPGAs contain a multitude of complex security features, but there are three main features present in all modern FPGAs that are relevant to the vulnerability described in this paper:

- Readback Prevention: There are many reasons why a hardware designer might want to prevent others from accessing their designs. Firstly, it generally very expensive and time-consuming to develop such designs, so it is important to protect the FPGA configuration from competitors or adversaries. Secondly, FPGA designs are often used in defense or aerospace applications where they might contain classified technology that could be stolen or replicated. In order to prevent this, most modern FPGAs forbid the configuration from being read back from the device. All recent Altera FPGAs prevent readback under any circumstances, but some Xilinx FPGAs allow the creator of the bitstream to determine whether or not readback of that bitstream will be allowed. Although this ability to prevent readback is invaluable for protecting the hardware designs, it also prevents a user or application from verifying the configuration of the FPGA.

- Bitstream Encryption: Since the configuration of the FPGA is volatile, the bitstream must be stored on another device and transferred to the FPGA once

it is powered up. Typically, it is stored in an external flash memory chip which is read by the FPGA on boot. Although the configuration can't be read directly from the FPGA, an adversary can easily read the external flash chip or intercept the data while it is transferred to the FPGA. To combat this attack, modern FPGAs utilize bitstream encryption. In order to use this feature, the FPGA must be loaded with a cryptographic key in a secure facility. The hardware designer can then encrypt the bitstream before storing it in the flash memory so that the FPGA can use its key to internally decrypt the bitstream as it is loaded. If an adversary intercepts this bitstream, they will be unable to decrypt it and access the design. In addition, since the key loaded into the FPGA is unique, the intercepted bitstream cannot be used to clone the design into another FPGA.

- Proprietary Bitstream Formats: Even if bitstream encryption is not used, a tampering attack is difficult to perform because the format of an FPGAs bitstream is proprietary and confidential. Although some bitstream formats have been reverse engineered for older devices [1], researchers have not been able to decode the formats for recent FPGAs. Because of this, it is very difficult for an adversary to manipulate a bitstream to make decisive changes to the FPGA's configuration, as any arbitrary changes will likely cause the bitstream format to be invalid, thereby causing the configuration to fail.

3.3 FPGAs as Accelerators

In this paper, we specifically discuss the use of FPGAs as accelerators, and we define an accelerator as an additional computing device which works in tandem with a general-purpose CPU. In recent years, hardware accelerators have become more prevalent in devices ranging from mobile phones to cloud servers. These accelerators enable the CPU to offload tasks to another hardware device which is better optimized for that task. A common example is the general purpose Graphics Processing Unit

(GPU) which is widely used to accelerate algorithms that can benefit from a large number of parallel cores. However, there is a trade-off between the level of optimization and the domain specificity. For instance, there are several examples of machine learning accelerators, from Google's Tensor Processing Unit, to the Movidius Vision Processing Unit. These accelerators benefit massively from heavy optimization, but are limited to performing a specific subset of machine learning algorithms.

Thanks to increases in size and effiency, FPGAs have become a more attractive option for hardware acceleration. The reconfigurable nature of FPGAs eliminates the trade-off between optimization and domain specificity. For this reason, FPGAs have appeared in cloud environments such as Amazon Web Services (AWS) EC2 F1 instance and embedded SoCs like the Altera Cyclone V. The former example uses a Xilinx UltraScale+ FPGA in a cloud server with a PCIe connection, similar to a more traditional GPU accelerator. The latter example is a single chip containing two ARM Cortex-A9 cores linked to an Altera FPGA. Some FPGA vendors have enabled their devices to be used with OpenCL, a framework designed for programming heterogeneous systems like those containing CPUs and GPUs. Since FPGAs require a hardware design for programming, the vendors have created High-Level Synthesis software for converting the C-like OpenCL kernel code to a hardware design. These efforts have enabled FPGAs to perform in an environment similar to that of the GPU.

An accelerator is used to supplement a CPU, but generally has a different architecture and discrete memory hierarchy. Because of this, there are some extra steps required to utilize them. A heterogeneous application is divided into a "host code" and "accelerator code", with the host code containing parts of the application which run on the CPU, and the accelerator code containing the parts to be offloaded to the accelerator. In OpenCL and CUDA, the accelerator code is also called the kernel. Because the microarchitecture and instruction set generally differs from that of the CPU, the accelerator code must be compiled separately from the CPU code using a specialized compiler. In a typical OpenCL program, the accelerator code is compiled with a vendor-specific compiler by the CPU "host" code at runtime. This runtime

compilation ensures that the accelerator code will run on the accelerator regardless of the specific model or architecture of the accelerator being used. In addition to compiling the kernel, the host application must allocate memory and transfer the data required by the accelerator code. Once the accelerator code has completed, the host code can read the results back into the CPU memory.

Using an FPGA accelerator is similar to a GPU accelerator but with one major difference: the FPGA application is determined by a hardware design instead of a software application, and the FPGA must be configured with this design before the host application utilizes it. For an FPGA accelerator, there are three ways of doing this:

- Kernel Module: Control registers are mapped to the operating system which can be accessed by a kernel module. The kernel module serves as an interface between the user space of the operating system and the FPGA hardware so that a user application can control the configuration. The user has the option of configuring the FPGA manually from the terminal or writing an application which automatically configures the FPGA during runtime. Once configured, host applications can utilize the logic implemented by the FPGA. The FPGA does not need to be reconfigured unless the system is rebooted or the logic on the FPGA needs to be updated for a different application.

- JTAG Programmer: It is also possible to configure the FPGA using an external device. An external programmer, such as the Altera ByteBlaster, can send the bitstream to the FPGA through the JTAG port. Alternatively, the bitstream can be automatically loaded during boot from a flash memory chip. Once the FPGA is configured from one of these devices, the CPU can utilize the logic implemented by the FPGA.

- OpenCL for FPGA: OpenCL is a framework designed for creating applications which run on heterogeneous systems. OpenCL is supported on a wide range of devices from by many vendors including Nvidia, AMD, Intel, and ARM.

This widespread compatibility combined with the heterogeneous programming model makes OpenCL a natural choice for enabling high-level programming on FPGAs. Several FPGA vendors have made OpenCL available for use with their FPGAs, allowing the devices to be used in a similar fashion to a GPU accelerator. Using a method called High-Level Synthesis (HLS), the OpenCL kernel containing the accelerator code can be translated to a hardware design for use with the OpenCL host code. Then when the host code is executed, the FPGA is configured with this bitstream by the OpenCL runtime (which utilizes a kernel module), and the CPU can then send tasks to it.

4. FPGA ACCELERATOR VULNERABILITY

In this section we explain how the reconfigurable nature of the FPGA creates a security vulnerability which is not present in other accelerators. We also describe an attack that can be performed on an application running on an SoC that contains an FPGA. In describing the attack, we refer to two agents: the "user", who is trying to run a legitimate application on the device, and the "adversary" who attempts to corrupt this application.

4.1 Attack Model

An FPGA can be configured at any time, and there is no lock to prevent the FPGA from being configured. Because of this, it is possible for an adversary to reconfigure the FPGA before or during the execution of an application. By configuring the FPGA with a modified bitstream, the adversary can alter the functionality of the program. By making random changes, it is possible to cause the application to crash, but by making more deliberate changes, the adversary can cause the FPGA to leak data or return incorrect results. There are two types of applications which can run on an FPGA accelerator: an HDL application, and an OpenCL application.

4.1.1 Case 1: HDL Application

This type of application uses low-level C code that interfaces with the FPGA using hardware memory addresses that correspond to data buses. In this type of application, the user must manually configure the FPGA with a hardware design that was created using traditional HDL methods in a language such as Verilog or VHDL. This hardware design is compiled to a bitstream, which is a file that can be

used to configure the FPGA. The user can use this bitstream to configure the FPGA through the operating system or through the JTAG port. Once the FPGA has been configured, the C code, running on the CPU, can utilize the design by interfacing with the FPGA through the mapped memory addresses. The development and runtime of an HDL application are illustrated in Figure 4.1.

The goal of the adversary is to reconfigure the FPGA after the user has configured the FPGA, but before the application ends. Like the user, the adversary also has the option to configure the FPGA through the operating system or through the JTAG port. Since it is impossible for the user or the application to verify the FPGA configuration, it would be difficult to determine whether an attack has succeeded unless the application crashed, started behaving erratically, or returned obviously incorrect results. However, the adversary can easily avoid these by making careful changes to the hardware design.

Before performing the attack, the adversary first needs to obtain a some information about the application. There are several ways to do this:

- If the adversary can obtain the original hardware design files (e.g. Verilog or VHDL code) then they can easily modify the design and recompile it. This is the most powerful method since the adversary would be able to change any aspect of the design. With very little effort, the design could be modified to corrupt the results of the application. The design could also be modified to leak sensitive data from the application.

- If the adversary has access to the C code, it is possible to reverse engineer a hardware design which can interface with the application without crashing. In this type of application, the C code would uses specific memory adresses for transferring data, checking status, etc. Because these addresses are configurable within the hardware design, two different bitstreams would likely not be interchangeable. In order to create a corrupt bitstream that would not simply crash the application, the attacker would need to ensure that the input

and output behavior of the new hardware design closely matches that of the original. This would be a difficult attack to perform since it would require a good understanding of the original application and extensive hardware design experience.

- The adversary may be able to obtain a previous version of a legitimate bitstream. One of the benefits of FPGAs is that, unlike other accelerators, their hardware can be updated over time. If a bug is discovered in a hardware design, then the programmer can fix the bug and update the bitstream. However, an adversary could use this attack to revert back to the older bitstream in order to re-enable the bug.

Since the FPGA does not lose its configurationg unless the chip is rebooted, the CPU application can be executed repeatedly without reconfiguring the FPGA. Since the configuration process takes several seconds, the user would likely try to avoid reconfiguration during each run of the application. To accomplish this, the user could have the FPGA configured during or shortly after the system boots. As such, the adversary has a wide window of opportunity to perform the attack; if the FPGA is reconfigured with a corrupted bitstream, then the application would be compromised until the system is rebooted.

There are some limitations to this attack model. For instance, in order to reconfigure the FPGA through the JTAG port, the adversary needs physical access to the FPGA and a JTAG programmer. On the other hand, to reconfigure the FPGA through software, the adversary would need the correct permissions within the operating system.

Case 2: OpenCL Application

The second type of application uses OpenCL to offload computation from the CPU to the FPGA. The development and execution of an OpenCL application on an FPGA accelerator is fundamentally different than that of an HDL application.

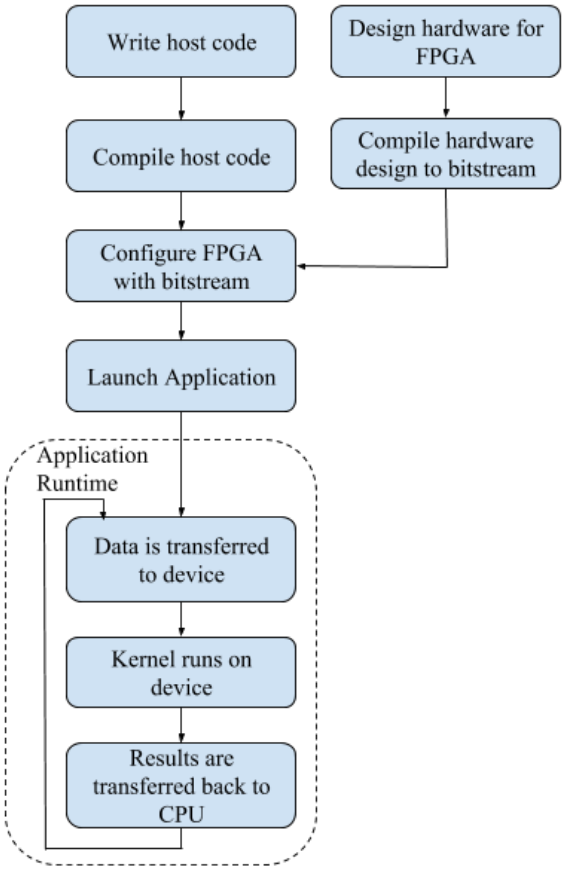

Fig. 4.1. Illustration of the development and runtime of a typical CPU+FPGA application. In this case, the FPGA is configured before the application launches with a bitstream created through traditional hardware design methods.

Instead of creating a hardware design using Verilog or VHDL, the programmer creates an "OpenCL kernel" which is written in a C-like language and compiled with a vendor-specific compiler. The result of the compilation process is a bitstream that can be used to reconfigure the FPGA through the OpenCL framework. Since the translation from C code to hardware design is handled by the compiler, the programmer does not need to do any hardware design to create an OpenCL application for an FPGA. The programmer also creates a host application that runs on the CPU. The host application loads the bitstream, configures the FPGA, transfers any data that is necessary, and then sends tasks to the FPGA. The configuration, data transfer, and

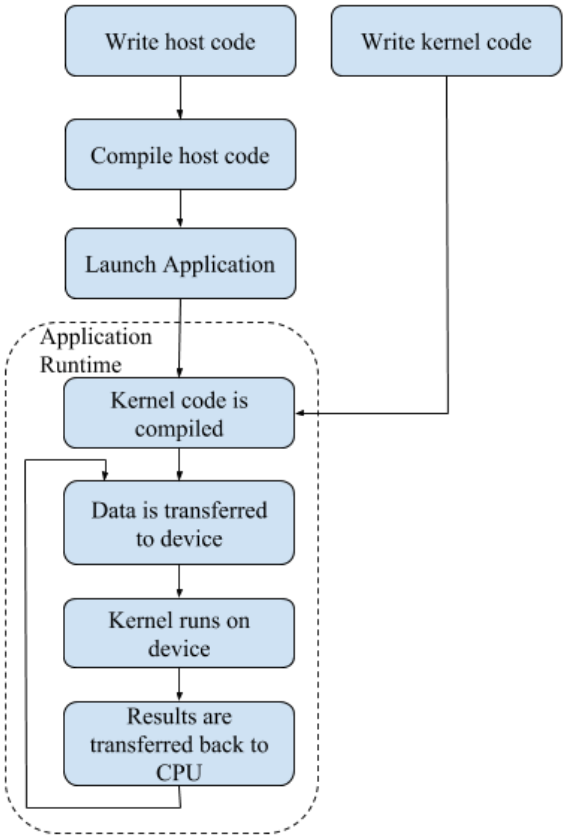

Fig. 4.2. Illustration of the development and runtime of a typical OpenCL application. Often the kernel is compiled during runtime to enable the kernel to run on a variety of accelerator architectures.

execution are all performed by standard OpenCL functions. Examples of OpenCL host code and kernel code can be found in Figures 4.1.1 and 4.1.1 respectively. The Figures 4.1.1 and 4.1.1 illustrate the development and runtime of a typical OpenCL application and an OpenCL application for an FPGA.

Many OpenCL programs can be run on an Altera or Xilinx FPGA with a few modifications. While OpenCL applications designed for GPUs or other non-configurable devices generally compile the kernel during runtime of the host application, a kernel for an FPGA must be compiled in advance. The process of compiling a bitstream from high-level OpenCL code is complex and takes anywhere from several minutes to several hours depending on the complexity of the code. Because of this, the pro-

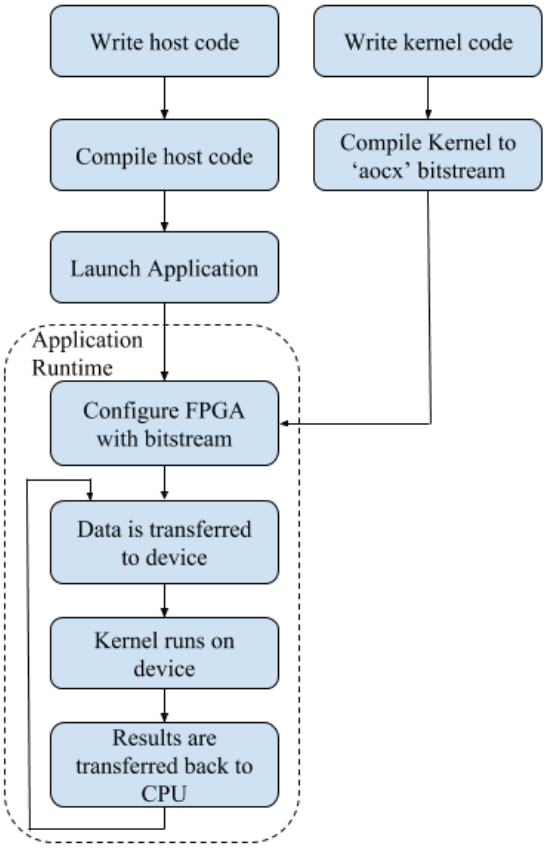

Fig. 4.3. Illustration of the development and runtime of an OpenCL application for an FPGA. The compilation of an FPGA kernel takes much longer than the compilation of kernels for GPUs or other accelerators, so the compilation is performed offline. The FPGA is configured during the runtime of the application, and multiple batches of data can be processed on the FPGA without reconfiguration.

grammer must compile the kernel ahead of time using a vendor-specific compiler, and then modify the host code to load the bitstream instead of the kernel source code.

The configuration process is controlled by the OpenCL implementation provided by the FPGA vendor. In the host code, the configuration is implemented as part of a standard OpenCL function that utilizes a kernel module to interface with the FPGA. The OpenCL runtime prevents multiple applications from using an accelerator simultaneously in order to prevent applications from interfering with each other. For GPUs and other non-configurable accelerators, the runtime simply has to prevent

a second application from running code or transferring data to or from the device. With an FPGA accelerator, the OpenCL runtime also must prevent a second OpenCL application from configuring the FPGA while an OpenCL application is running. However, it is still possible for an adversary to bypass the OpenCL runtime and configure the FPGA directly through a driver or through the JTAG port.

The FPGA configuration process requires a non-negigible amount of time, generally on the order of seconds. To prevent this latency from causing a performance issues, a typical OpenCL application will configure the FPGA at the start of the host application, and avoid reconfiguration if possible. Therefore, if an adversary reconfigures the FPGA after the application has configured it, then the application will not correct the problem. Instead, the application will continue to run in its compromised state.

4.2 Our Equipment

For our experiments, we used the Terasic DE10-Standard, which is a development kit centered around a Cyclone V SoC. This SoC contains two ARM Cortex-A9 cores connected to a reconfigurable FPGA fabric. The board contains many connectors including an ethernet port, a port for serial communication, and an Altera USB Blaster II which interfaces with the SoC through a JTAG connection. Terasic provides an installation image for running Angstrom Linux with kernel 3.10.31-ltsi on the two ARM cores. This image also contains a driver which can be used to reconfigure the FPGA and the Altera OpenCL runtime for use with the FPGA.

4.2.1 Cyclone V SoC

The Cyclone V SoC is divided into two distinct parts: the Hard Processor System (HPS) which contains the ARM cores, and the FPGA portion. The HPS and FPGA can communicate through the FPGA Manager and the three Advanced eXtensible Interface (AXI) buses. The FPGA Manager is a control block within the HPS which is

primarily responsible for configuring the FPGA. A set of control and status registers is accessible through mapped hardware addresses in the CPU. It is important to note that the configuration of the FPGA is always performed by the FPGA Manager regardless of whether the configuration is initiated by the HPS or an external JTAG programmer. The internal layout of the Cyclone V SoC is illustrated in Figure 4.6.

In addition to the FPGA Manager, the HPS and FPGA can use a set of AXI bridges to communicate directly. The HPS-to-FPGA bridge and the FPGA-to-HPS bridge are used for transferring data to and from the FPGA fabric respectively. These two bridges are configurable to support 32, 64, or 128 bit data widths. The two portions of the SoC can also communicate through the Lightweight HPS-to-FPGA AXI bridge, which is a lower bandwidth bridge that is primarily used for accessing control and status registers for peripherals in the FPGA fabric. The HPS and FPGA fabric both have access to external SDRAM memory through the SDRAM controller within the HPS, however each device has a dedicated memory region which is not accessible by the other device. In order to provide the FPGA with access to the SDRAM controller, there is an FPGA-to-HPS SDRAM interface which supports AXI and Avalon Memory-Mapped interface standards.

Although we performed our experiments on the Cyclone V SoC, Altera also produced an Arria V SoC and Stratix V SoC which utilize the same overall architecture as the Cyclone V SoC. The primary difference between the three lines of devices is the size of the FPGA fabric. The FPGA fabric in these devices contains resources such as Logic Elements, Adaptive Logic Modules, DSP blocks and other components that are used to implement the hardware specified by the bitstream. The Arria V and Stratix V devices contain more resources, allowing them to fit larger, more complex designs. Despite additional resources in the larger devices, the HPS and FPGA portions of the SoC are still connected by the FPGA Manager and AXI bridges, and the configuration process is the same. As such, these devices are also vulnerable to our attack model.

4.3 Cyclone V SoC Vulnerability

In this section, we describe the security vulnerability in the context of the Cyclone V SoC. We have performed this attack on our Terasic DE10 and describe solutions to this vulnerability in the next section.

4.3.1 Reprogramming the FPGA

The variety of methods for programming the FPGA combined with the lack of readback ability makes it impossible for an application to verify that the hardware design currently implemented by the FPGA is the expected design. It is possible for a user or application to program the FPGA using a driver which is provided with the operating system. The driver included with the board creates a file at /dev/fpga0 in the operating system, and if a user or application writes an FPGA bitstream to this file, the driver will utilize the FPGA Manager to configure the FPGA with the bitstream. Alternatively, a user can connect a JTAG programmer to the chip and write the bitstream to the FPGA using Quartus software running on a separate computer. Despite the use of an external programmer, the configuration of the FPGA is still performed by the FPGA Manager block in the HPS. During the reconfiguration process, applications utilizing the FPGA cannot detect this change.

The naive solution to this problem would be to continuously check the configuration of the FPGA to verify that it has not changed unexpectedly. However, as previously explained, readback of the configuration is disabled in order to prevent the cloning or reverse engineering of hardware designs. Even if this feature was available, it would not be practical, since the FPGA bitstream tends to be on the order of several megabytes; to continuously read this bitstream back to the CPU and compare it to the expected configuration would be very costly. As a result of these limitations, it impossible to design a secure application using the provided driver.

4.4 Performing the Attack

We successfully performed the attack on 5 different applications: 2 HDL applications, and 3 OpenCL applications.

4.4.1 Attacking an HDL Application

In order to demonstrate the attack, we use an open source application called MD5 Cracker [2]. The application was designed for the Cyclone V SoC and uses the FPGA portion of the SoC to perform an MD5 hash function in hardware. The application uses the AXI bridges to send data to the FPGA memory and read the resulting hash back to the CPU memory when the computation has completed. In order to use the application, the bitstream must by generated by Altera's Quartus software using the hardware design files provided with the application. Once the bitstream is obtained, the user must configure the FPGA from the HPS or an external JTAG programmer. After the FPGA is configured, the user can launch the host code on the CPU. Although the original application was designed to calculate several MD5 hashes in order to demonstrate the performance of the FPGA, we modified the application to accept an input file and output the hash. As a result, the application worked similarly to the *md5sum* command in Linux, so we can easily check the correctness of the FPGA's calculation.

The MD5 algorithm begins by initializing four 32-bit variables that make up the 128-bit output hash. Each of the variables keeps a running sum of values calculated in the algorithm, and they are initialized with four very specific numbers. Changing any of these numbers by even a single bit drastically changes the output hash to an erroneous value. In addition, these initial values are hard-coded into the hardware design provided with the MD5 Cracker project.

As per our attack model, we have two agents: the user and the adversary. The user is trying to use the FPGA to calculate the MD5 hash of input files, and the adversary is trying to corrupt the user's results without their knowledge. The user has

the option of configuring the FPGA with software on the HPS or using an external JTAG programmer, and their chosen method of programming does not affect the operation of the application in any way. Once the user has configured the FPGA, the adversary can reconfigure the FPGA using a corrupted bitstream. In our case, this corrupted hardware design was identical to the original, except one of the initial values for the MD5 algorithm which was modified by one bit. Regardless of how the user configured the FPGA, the adversary has the option to use either the kernel module or JTAG port. The configuration does not provide any warning to the MD5cracker application. Therefore, after the adversary reconfigures the FPGA, the user's results will be incorrect, and the only way to discover that the results are incorrect would be to cross-check with another method, such as a software implementation of the MD5 algorithm. However, it would be foolish to check the results in software since this would negate any performance benefit provided by the hardware implementation of the algorithm. As a result, the user has no efficient way to ensure the correct operation of the FPGA.

In addition to the MD5cracker application, we tested the attack on a very simple program provided with the DE10 board. The application demonstrates the usage of on-board LEDs that are connected directly to FPGA pins. The hardware design provided with this application effectively links the LEDs to the CPU to enable software on the CPU to control them. As with the MD5cracker application, the user must manually configure the FPGA before running the host code on the CPU. However, by modifying the connections of registers in the hardware design, the adversary can create a hardware design that alters the pattern in which the LEDs are blinked. Although this is a very simple application, it demonstrates the ability of our attack model to alter the functionality of hardware peripherals like networking ports that may be directly connected to the FPGA.

4.4.2 Attacking an OpenCL Application

Altera's OpenCL SDK supports two different methods of programming the FPGA. Both of these methods are performed through a driver in the operating system. Since the OpenCL runtime prevents an application from reconfiguring the FPGA during the execution of another OpenCL program, it is impossible to perform the attack using the officially supported programming methods. However, we found a third method that allows the adversary to reconfigure the FPGA with an OpenCL kernel using a driver in the operating system or the JTAG port. As a result, we have three methods for configuring the FPGA for an OpenCL application:

- OpenCL Host Code: The FPGA can be configured using the standard OpenCL function "clCreateProgramFromBinary()" within an OpenCL host application. This function has been modified by Altera to configure the FPGA through an operating system driver. This function typically runs at the beginning of the host application, and is the default method of configuration for an OpenCL program.

- Altera OpenCL SDK: It is also possible to configure the FPGA from the command line using Altera's OpenCL SDK. The SDK includes an executable file called "aocl". This program allows a user to configure the FPGA using a bitstream in the "AOCX" format. This can be done when an OpenCL application is not currently running.

- Create an HDL Design: When compiling the hardware design for an HDL application, Quartus produces a bitstream file with an "sof" extension. However, OpenCL applications use a different filetype with the extension "aocx". Because the formats of these files are confidential, we don't know exactly what the difference is, but we do know that the formats are not interchangeable. Therefore, the adversary is faced with two problems: (1) The OpenCL runtime does not allow the adversary to reconfigure the FPGA during the execution of an

OpenCL application. (2) The adversary cannot use the aocx file to reconfigure the FPGA using the operating system driver or the JTAG port. However, we found that during the compilation process, Altera's OpenCL compiler automatically generates an HDL design and then uses that design to create the "aocx" bitstream. These HDL design files can be loaded into Quartus in order to generate the typical "sof" or "rbf" filetype which are respectively used to configure the FPGA externally using a JTAG programmer and through software using a driver. The functionality of this hardware design is identical to that of the "aocx" bitstream.

Using the method described above, the adversary can create hardware design files from an OpenCL kernel code. With this ability, it is very simple for the adversary to modify the C-like kernel code and compile this kernel into a corrupted bitstream. This corrupted bitstream can be used to configure the FPGA with the OpenCL kernel while bypassing the FPGA.

We successfully performed this attack on three different OpenCL applications:

- Matrix Multiplication: This application creates two matrices of random floating point values and transfers them to the FPGA where they are multiplied. The results are then read back to the FPGA and checked for correctness.

- Vector Addition: This application creates two vectors of random floating point values and transfers them to the FPGA to perform element-wise addition. This application was provided as an example with the Terasic board.

- Nearest Neighbor: In order to demonstrate the attack on a more realistic application, we used the Nearest Neighbor benchmark from the Rodinia Benchmark Suite [3]. This application uses hurricane data to compute the nearest location to a certain location for a number of hurricanes. Of the Rodinia benchmarks, we selected this one because it has only one kernel. Many of the Rodinia applications utilize multiple complex kernels which do not fit on our Cyclone V FPGA.

To test the attack, we modified the applications so that they configure the FPGA and then sleep for 5 seconds. After this, the application repeatedly executes the kernel on the FPGA and the sleeps again for 5 seconds. This pattern is meant to emulate the case where an application intermittently utilizes the FPGA for acceleration of a specific task without reconfiguring it each time. It is not uncommon for OpenCL applications to process data in batches due to the limited amount of memory found on many accelerators. It is also not uncommon for an application to perform some processing on the CPU before offloading the data to the accelerator. This pause gives the adversary some time to configure the FPGA with the corrupted bitstream.

For each application, we modified the OpenCL kernel and recompiled it in order to obtain the corrupted bitstream used in the attack. In each case we modified the kernel as little as possible to yield incorrect results by either changing a hardcoded value or incrementing the variables. In all cases, the attack successfully corrupted the results without crashing the application.

4.5 Testing Degrees of Change

In order to test how much the OpenCL kernel could change before the whole application crashed, we created 6 different versions of the Matrix Multiplication kernel each with a varying degree of change:

- Version 1: Original, unmodified matrix multiplication kernel
- Version 2: This is the kernel we previously used to test the attack. In the output matrix, each value is incremented by 1.
- Version 3: This version is almost entirely unchanged except there is an extra argument added to the kernel function definition.
- Version 4: In this version, the algorithm is changed to calculate the Hadamard product (element-wise multiplication) instead of matrix multiplication.

- Version 5: In this version, the algorithm is changed to perform 2-D convolution on one of the input matrices instead of matrix multiplication.

- Version 6: This version has almost everything removed from the code. It consists of a function name with no arguments and no operations in the function. It is the bare minimum amount of code that can be compiled by the Altera OpenCL compiler.

Remarkably, when we performed the attack using these kernels, none of them caused a crash. Regardless of the degree of change, each kernel simply changed the results of the application. Also, for the kernels that performed a completely different algorithm (Versions 4 and 5), the results of the new algorithm were correct. In other words, it is possible to completely hijack the application to perform any arbitrary function. Considering even Version 6 did not cause a crash despite the kernel taking no arguments and performing no operations, there is presumably no limit to the amount of change that can be performed to the OpenCL kernel.

```
// Get platforms
cl_platform_id platform_id;
clGetPlatformIDs(&platform_id);

// Get device ID
cl_device_id device_id;
clGetDeviceIDs(platform_id, &device_id);

// Create context
cl_context context = clCreateContext(&device_id);

// Create command queue
cl_command_queue commands = clCreateCommandQueue(context, device_id);

// Load pre-compiled kernel
char* kernel_bin = LoadKernel("matmul.aocx");

// Create kernel object
cl_program program = clCreateProgramWithBinary(context, &device_id,
    &kernel_bin);
err = clBuildProgram(program);
cl_kernel kernel = clCreateKernel(program, "matrixMul");

// Create buffers
A_dev = clCreateBuffer(context, A_size, A_host);
B_dev = clCreateBuffer(context, B_size, B_host);
C_dev = clCreateBuffer(context, C_size);

// Set arguments of kernel function
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&C_dev);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&A_dev);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&B_dev);
err |= clSetKernelArg(kernel, 3, sizeof(int), (void *)&A_width);
err |= clSetKernelArg(kernel, 4, sizeof(int), (void *)&C_width);

// Begin execution on device
err = clEnqueueNDRangeKernel(commands, kernel);
```

Fig. 4.4. This figure contains pseudocode which illustrates the structure of OpenCL host code which is designed to use an FPGA accelerator. With the exception of LoadKernel(), the functions names are all valid OpenCL functions, however the arguments have been simplified in order to best illustrate the purpose of each function. In the Altera OpenCL runtime, the clBuildProgram() function has been modified to configure the FPGA with the loaded binary.

```
__kernel void matrixMul(__global float* restrict C, __global float*
    restrict A, __global float* restrict B, int wA, int wB)
{
  int tx = get_global_id(0);
  int ty = get_global_id(1);
  int k = 0;
  float value = 0;

  for (k = 0; k < wA; ++k) {
    value += A[ty * wA + k] * B[k * wB + tx];
  }

  C[ty * wA + tx] = value;
}
```

Fig. 4.5. This figure contains valid OpenCL kernel code to perform matrix multiplication. To run this code on an FPGA, it must be compiled in advance by a compiler created by the FPGA vendor. Instead of loading and compiling the kernel code, the application loads the precompiled bitstream and configures the FPGA.

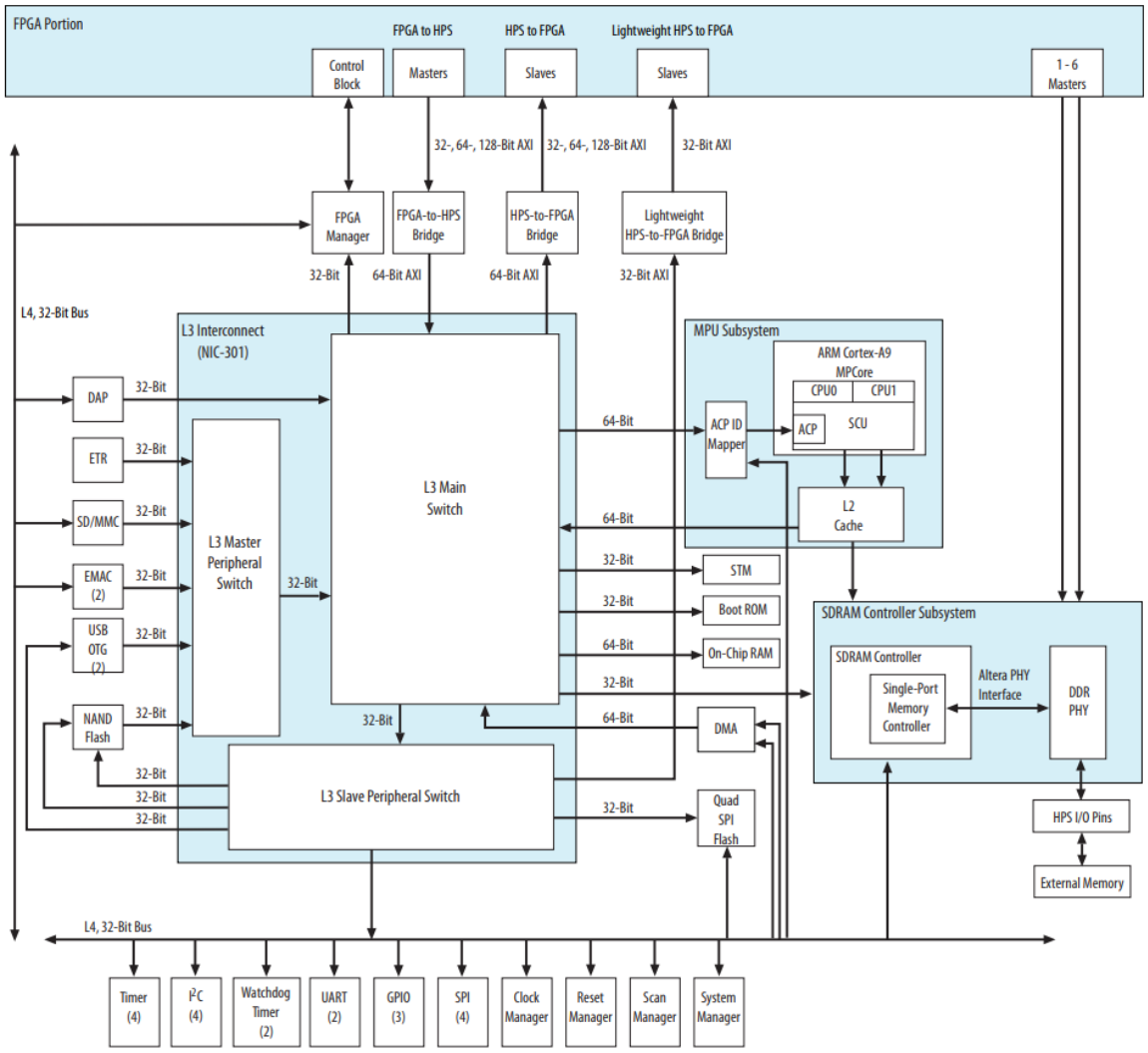

Fig. 4.6. Layout of the Cyclone V chip showing connections between HPS subsystems and the FPGA portion within the SoC. This figure is taken from the Cyclone V Device Handbook.

5. SOLUTION

In this section, we put forth two solutions for eliminating the vulnerability described in this paper. The first is a hypothetical hardware modification which would provide a very robust method for verifying the current configuration of the FPGA. The second solution is a software implementation of this hardware change which demonstrates a proof-of-concept for the hardware design. In addition, the software solution can be used to greatly improve the security of the system, but with some additional limitations.

5.1 Hardware Solution

In the previous section, we describe in detail the process of configuring the FPGA portion of the Cyclone V SoC. We explain that the FPGA Manager portion of the HPS always controls the configuration of the FPGA regardless of whether the configuration is initiated by software running on the CPU or by an external JTAG programmer. In addition, the FPGA Manager contains registers that can be accessed by software running in the operating system on the CPU. Because of this, we conclude that if the FPGA Manager can keep a record of the FPGA configuration, then an application should be able to check these records in order to verify the configuration. Since there is no way to configure the FPGA while bypassing the FPGA Manager, the records will always be accurate.

The records kept by the FPGA Manager must be able to provide information on the state of the FPGA without providing details about the hardware design, since this would create other security issues. In order to do this, we suggest that the FPGA Manager incorporate an additional hardware device to create a hash of the bitstream. During the configuration process, the bitstream data streams into the FPGA Manager

which decodes the data and sends it to the SRAM registers in the FPGA fabric. A hashing device could break the data into blocks with a fixed number of bits and use a hashing algorithm to calculate the hash of the block. The hash would be stored in a register, and the hash of each successive block would be XORed with the contents of the register. After the configuration has completed, the register would contain a hash for the bitstream which could be accessed by applications running on the CPU. Since both the contents of this register and the FPGA configuration could only be modified by the FPGA Manager, the hash value would always contain the hash of the current configuration.

The chosen hash algorithm would depend on the level of security desired and the difficulty of performing a collision where a corrupted bitstream has a matching hash value and is also a valid bitstream. Since FPGA manufacturers keep the format of their bitstreams confidential, it is nearly impossible to modify the bits of a bitstream to make a decisive change. An adversary could randomly modify bits in the bitstream, but this is likely to create an invalid bitstream which will be discovered by the FPGA Manager during the configuration process, leading to a failed configuration. Alternatively, an attack could try to find a hash collision by modifying the source files for the hardware design and generating a new bitstream. However, even with a powerful computer, generating a single bitstream can take anywhere from several minutes to several hours depending on the complexity of the design, so trying to brute-force a hash collision is impractical. For these reasons, hashing the bitstream within the FPGA Manager would provide a robust solution to this vulnerability.

5.2 Software Implementation

In order to demonstrate our solution as a proof of concept, we implemented the solution as a kernel module within the operating system. Since the FPGA has registers that are mapped to hardware memory addresses accessible by the CPU, the FPGA

configuration can be controlled by software within the operating system. Our kernel module uses these control and status registers to configure the FPGA.

5.2.1 Description of the Kernel Module

Because the default driver is closed source, we created a new driver with almost identical operation. Once loaded, the driver maps the necessary physical memory addresses into variables which can be accessed by various functions within the driver. The driver also creates a character device which is located at /dev/fpga_config in the operating system. In order to program the FPGA, a user uses the *dd* command to write an FPGA bitstream to the character device.

At any point, the FPGA is in one of five phases: Power-up phase, Reset phase, Configuration phase, Initialization phase, or User mode. When the SoC is turned on, the FPGA defaults to the Power-up phase. When the character device is opened, the kernel module modifies values in the FPGA Manager's registers to put the FPGA into Reset phase and then the Configuration phase. In addition, a *count* variable is incremented each time the character device is opened in order to keep track of the number of attempted configurations, and a *hash* variable is set to 0. At this point, the driver starts reading the bitstream in blocks of 32 bits. For each block, the 32-bit value is written to a *data* register within the FPGA Manager which sends this data to the FPGA fabric for configuration. Besides writing the value to the *data* register, the 32-bit block is cast as an unsigned 32-bit integer and added to the *hash* variable. It should be noted that a basic summing operation was used to simplify the implementation, but an advanced hash algorithm such as SHA-1 could be used to prevent the likelihood of a successful collision attack. However, this basic summing procedure still deomonstrates the solution since, despite it's simplicity, a collision is relatively rare. Once the FPGA Manager detects the end of the bitstream, the FPGA transitions into the Initialization phase and then User mode. The User mode

indicates that the configuration was successful and the FPGA fabric has implemented the logic described by the bitstream.

Once the configuration is complete and the character device file has been closed, a register in the FPGA Manager is modified to disallow access to the FPGA Manager through the JTAG port. As a result, the FPGA can only be configured by users with access to the character device file in the operating system. In addition, the *hash* and *count* variables are written to a file at /proc/fpga_config. By reading this file, a user or application can check the integer value created by summing the 32-bit blocks of the bitstream as well as the number of times reconfiguration has been attempted since the kernel module was loaded. If an adversary manages to reconfigure the FPGA, the sum will change and the count will be incremented. If an application is designed to verify these values in the /proc/fpga_config file, it will be invulnerable to such an attack. The /proc file would allow an application to detect the attack and automatically reconfigure the FPGA with the correct bitstream which would minimize the interruption. A high number of reprogrammings would also alert the system administrator of a malicious user. The best case scenario for the adversary would be to temporarily disrupt the application until the FPGA is reconfigured.

5.2.2 Performance Metrics

In order to ensure our custom driver did not introduce a significant overhead, we took three measurements that demonstrate the speed of configuration and validation. To demonstrate that the checksum did not significantly increase the time to configure the FPGA, we measured the configuration time using our custom driver with and without the summing operation. We found that the summing operation added less than 1% to the configuration time.

To increase the security of the driver, a user would most likely wish to use a cryptographic hashing algorithm such as SHA-1. However computing the hash of a typical bitstream took roughly 50 milliseconds on the low-power embedded CPU

Table 5.1.
Peformance measurements of software solution.

	Average	Stdev
Config Time w/ checksum	1.614 s	0.0152 s
Config Time w/o checksum	1.601 s	0.0179 s
SHA-1 Hash Time	0.0505 s	0.002179 s
Verification Time	70.15 μs	86.47 μs

within the Cyclone V SoC. Substituting a SHA-1 hash for the checksum would raise the overhead to about 3.2%. Since a typical application would only reconfigure the FPGA once per execution (or even once per boot), this overhead is negligible.

In order to secure an FPGA application using our driver, the application would need to verify the checksum and compare it with a known value. This would require the application to open the file at /proc/fpga_config, read the string, and compare it with a known value. The time to perform this process on our Cyclone V SoC is only 70 microseconds. A programmer has many options when it comes to using this value to secure the application. The application could poll the value at a certain interval in order to verify that the FPGA has not been tampered with. Alternatively, the application could read the checksum and reconfiguration count from the fpga_config file before and after receiving the computation results from the FPGA. Regardless of the verification method, an overhead of 70 μs should have a very minimal impact on an application.

Practicality of Software Solution

Although the kernel module was created to demonstrate a proof-of-concept for our hardware fix, the software implementation may be useful in practice with some limitations. In order for the driver's hash value to be accurate, the FPGA must always be configured using the driver. For this reason, it cannot be configured from the JTAG port. Fortunately, the FPGA Manager can disallow JTAG configuration, but this eliminates that option for any valid user that may wish to utilize this ability. In

addition, it is possible to bypass our driver and configure the FPGA using the original driver, but this would require the adversary to have permissions to the original driver's character device file which by default requires root access. It is possible to solve this problem within a trusted operating system with user groups. For instance, on our system, the custom character device file is owned by a group called *fpga_control*. A non-root user in this group would be able to configure the FPGA through our custom driver, but not through the original driver or any other method.

In its current form, the custom driver can secure the FPGA in the case of a trusted OS, but not an untrusted OS. Our software solution relies on the fact that the custom driver is the only method for a user to control the FPGA. However, if the OS itself is compromised, then the OS would have full control of the FPGA through its direct access to the FPGA Manager registers. If the communication between the CPU and FPGA Manager can be restricted to a secure enclave, such as Intel SGX or ARM TrustZone, then the driver can provide a secure solution even within an untrusted OS. The ARM cores within Altera's range of SoCs contain hardware support for ARM TrustZone, so this is a possibility.

6. CONCLUSION

While most work on FPGA security focuses on embedded systems, FPGAs are becoming more common in SoCs and cloud environments for hardware acceleration. Despite their newfound popularity, there is little work being done to address the security of FPGAs in these settings. We demonstrate an attack on an Altera Cyclone V SoC in which the FPGA is reconfigured during the execution of an application in order to corrupt the results. In order to prevent this attack, we propose a hardware modification that would calculate a cryptographic hash of the FPGA bitstream during configuration, allowing software to verify the configuration without extracting information about the IP on the FPGA. As a proof-of-concept, we created a kernel module that can configure the FPGA while calculating a checksum that can be verified by an application running on the SoC. The implementation of our solution by FPGA vendors would greatly improve the security of FPGA accelerators by eliminating this vulnerability.

REFERENCES

REFERENCES

[1] F. Benz, A. Seffrin, and S. A. Huss, "Bil: A tool-chain for bitstream reverse-engineering," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 735–738.

[2] H. Mao, "Md5 cracker," 2015. [Online]. Available: https://github.com/zhemao/md5cracker

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[4] J. Zhang and G. Qu, "A survey on security and trust of fpga-based systems," in *2014 International Conference on Field-Programmable Technology (FPT)*, Dec 2014, pp. 147–152.

[5] M. Zhao and G. E. Suh, "Fpga-based remote power side-channel attacks," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 229–244.

[6] N. Jacob, J. Heyszl, A. Zankl, C. Rolfes, and G. Sigl, "How to break secure boot on fpga socs through malicious hardware," in *Cryptographic Hardware and Embedded Systems – CHES 2017*, W. Fischer and N. Homma, Eds. Cham: Springer International Publishing, 2017, pp. 425–442.

[7] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "Confidentiality issues on a gpu in a virtualized environment," in *Financial Cryptography and Data Security*, N. Christin and R. Safavi-Naini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 119–135.

[8] R. D. Pietro, F. Lombardi, and A. Villani, "CUDA leaks: Information leakage in GPU architectures," *CoRR*, vol. abs/1305.7383, 2013. [Online]. Available: http://arxiv.org/abs/1305.7383