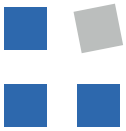


HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Embedded Secure Boot Test Suite – Thesis

A Generalized Approach in Assessing the Physical Security of Embedded Secure Boot

Bachelor's Thesis, Spring Term 2020

Departement of Computer Science
Hochschule für Technik Rapperswil (HSR)
`www.hsr.ch`

Author: Marco Reifler
Author: Marco Zollinger

Advisor: Prof. Stefan Richter, HSR

June 12, 2020

Abstract

The first and therefore most important stage in the secure boot process is the root of trust (ROT), a piece of proprietary hardware or firmware deeply embedded into the chip. If it is compromised, the rest of the boot process cannot be trusted. Implementations are however usually confidential, and its security cannot be verified independently. This project provides a test suite to analyze ROT firmware security in different processors. Most ROT firmware is permanently burned into the chip during production, which makes security vulnerabilities impossible to patch. Combined with the high execution privileges, they are a worthwhile target for attackers. The firmware memory locations in the chip are usually read-protected and the code cannot be extracted and reverse engineered for security auditing. Glitching attacks may temporarily override the read-out protection or coerce the chip into accepting unauthenticated software images. Alternatively, black-box fuzzing can be used to test the interfaces for vulnerabilities without knowledge about the inner workings of the firmware. A generalized hardware system has been designed to support experimenting with physical attacks like power analysis and glitching, which are difficult to perform on off-the-shelf hardware without extensive modifications. The design can be adapted for different target processors, as has been demonstrated by developing a custom board for the Zynq-7000 system-on-chip. The result is an experimentation board that provides interfaces for security testing and allows for a reliable laboratory setup. To support the experiments, a test suite consisting of guidelines, procedures and attack scripts has been elaborated. Control flow manipulation of user code running on the target platform through clock glitching has been successfully demonstrated. Further research is necessary to confirm that the ROT firmware is also susceptible to such manipulations, or if there are countermeasures in place.

Lay Summary

With the increasing digitalization of our world and daily lives, cyber security is more important than ever. Without countermeasures, attackers could transfer bank funds without authorization, steal sensitive information or sabotage critical infrastructure. Usually, important computing systems are running in secured server farms, protected from both physical and digital attacks with considerate effort. This is however not the case for many computer-controlled electronic systems from webcams, to computer networking equipment, to even cruise missiles. What all these systems have in common is that they are at some point deployed in the field and need to withstand attacks on their own, especially physical tampering. Some microchip manufacturers provide security measures against unauthorized access, sabotage and software manipulation, but their techniques are mostly proprietary and not open to independent security audits. This requires device manufacturers to trust them blindly, not knowing how effective these security measures actually are.

The goal of this thesis is to aid device manufacturers and researchers in assessing the security of microchips. To achieve this, a test suite with different techniques and attacks was put together. Tools to perform such experiments are already available, but this thesis specifically shows how to approach testing a new microchip. This includes finding viable attack opportunities and adapting the tools to perform the experiments on this chip. The process was demonstrated with an example microchip not previously supported by the commercial tools and a potential vulnerability was shown.

Contents

List of Figures

List of Tables

List of Listings

1 Management Summary

1.1 Introduction

This bachelor thesis is a follow-up to our last student research project "Secure Boot on Embedded Systems". A secure boot process on the Xilinx ZYNQ-7000 system-on-chip (SoC) was implemented according to the recommendations of the manufacturer. After an analysis of the available security options, the boot image was both encrypted and authenticated to ensure its authenticity and confidentiality. An in-depth security analysis on the chip itself, was however outside the scope of the project. The secure boot process is based on a chain of trust, where every stage relies on the security of the previous one. The first step in this chain, the root of trust (ROT), is therefore the most important one. If it is compromised, the rest of the secure boot process cannot be trusted. In most cases, the ROT is a piece of proprietary hardware or firmware deeply embedded into the chip. It is confidential and cannot be read out by the user. This requires users to put a lot of trust in the manufacturer and the chip, because its security cannot be independently verified.

1.2 Approach

The goal of this project is to provide a test suite to analyze the ROT firmware security in different processors and SoCs. Software vulnerabilities in user code can be fixed with security patches, but this is usually not possible with ROT firmware. They are burned into the chip during production and cannot be altered afterwards. Because of their high execution privileges and immutability, vulnerabilities in ROT firmware can be very valuable and dangerous. However, the firmware memory locations in the chip are usually read-protected and the code cannot be extracted, and reverse engineered by the user for security auditing. Glitching may provide means to temporarily manipulate the chip and override this read-out protection, or to coerce it to accept unauthenticated software images, if physical access to the device is possible. If the code cannot be extracted, black-box fuzzing methods can still be applied to test the interfaces for security vulnerabilities without knowing anything about the inner workings of the firmware.

1.3 Result

A generalized hardware system design has been elaborated from the requirements to support experiments with physical attacks like power analysis and voltage glitching or clock glitching. These experiments can be difficult to perform on off-the-shelf development boards without extensive modifications. The generalized design can be adapted and implemented for different target processors or SoCs. For this project, the process has been demonstrated by developing a custom board with the ZYNQ-7000 SoC. The result is an experimentation board that provides interfaces specifically engineered for security testing and allows for a clean laboratory setup

with superior reliability and repeatability. To support the experiments, a test suite has been elaborated. It consists of code analysis guidelines, and procedures and scripts to support power analysis, fault injection and fuzzing attacks. Instruction skipping and manipulation of the control flow of user code running on the target platform through clock glitching have been successfully demonstrated. Further research is needed to confirm that the root of trust firmware is also susceptible to such manipulations or if there are countermeasures in place.

1.4 Outlook

This test suite is built in a modular way so new tests can be easily added or existing ones expanded on. Most of the tests allow further automation for a development pipeline. The power analysis experiments should be performed on a live system and an attempt at decrypting secret information should be made with the purpose of verifying the results. Performing more sophisticated attacks, e.g. correlation power analysis, would only be necessary if differential power analysis does not show the expected results.
Since a clock glitch was successful, an attempt using this glitch can be made to bypass authentication. Different glitching methods should be added to allow testing of a wider range of devices.
The fuzzing software needs to be used on the target to check the target's robustness and to verify the fuzzer. Additional knowledge about system internals would enable other fuzzing strategies.

2 Introduction

2.1 Motivation

With the rapid rise in number and complexity of IoT devices, the demand for security has increased accordingly. The increasing value of intellectual property deployed in the field asks for better protection. In our semester project 2019 a secure boot solution was implemented for the Xilinx ZYNQ-7000 SoC. The tests performed on the system were basic enough as to not require any equipment. Considering the many successful attacks performed on embedded devices in recent years, we sought to further expand on the evaluation methods for embedded systems and include more of the modern hardware hacking methods.

2.2 Current Situation

New vulnerabilities in embedded systems are discovered frequently and often reveal grave security issues in established systems. Current secure boot implementations use encryption to hide the software partitions from prying eyes as well as authentication to prevent foreign code execution. What they often fail to consider or prevent are physical hacking methods. Unlike traditional computer systems and networks, embedded systems are often deployed in the field and physical access is impossible to prevent. This opens the door for previously less known attack methods. Side-channel attacks are used to extract secrets from cryptographic systems and glitching is used to manipulate a running system, allowing the manipulation of program data even without logical faults in the software. Measures to prevent or impede these attacks are difficult to implement and need to be considered from the very beginning of a system design process. With many of the security measures of microchip manufacturers being proprietary and inaccessible to independent parties, it is difficult for device manufacturers to evaluate them. The main contribution of this thesis is a method of testing such a system to make this process more accessible.

2.3 Assignment

This project seeks to develop methods of testing microchips against some of most common contemporary hacking methods used against embedded systems and compiling them into a test suite that should be adaptable to different systems with minimal effort. The tests are targeted towards the ZYNQ-7000 SoC by Xilinx and most of the experiments are performed on that system. For the purpose of experimentation, a custom evaluation board should be designed to create an optimal environment for the attacks included in the test suite with the goal of maximizing test performance and repeatability.

3 Related Work

This chapter surveys previous work in embedded security and attack vectors on embedded systems. A lot of the work focuses on a single method of attack to breach security on an embedded system.

3.1 Code Analysis

Code analysis is an extremely broad topic in computer science and an important part of software engineering. Understanding code analysis tools can help software engineers to create better software, but they can also help potential attackers find vulnerabilities. Automatic code inspection or analysis tools are separated into static and dynamic tools. Manual code inspection of code, line by line, still plays an important role today, especially in secure code.

Drew Buttner. The importance of manual secure code review. Mitre. Jan. 16, 2014. URL: `https://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/the-importance-of-manual-secure-code-review` **(visited on 03/17/2020)**
Manual code review is often used in addition to automated tools. Its costs can be prohibitive, and it is therefore often used only for critical code sections instead of whole software systems. Its biggest advantage is that it can find flaws that no other method can find.

P. Louridas. "Static code analysis". In: IEEE Software 23 (2006), pp. 58–61
This article is a bit older but still relevant, because it highlights the important aspects of static code analysis. Its use in software engineering for preventing vulnerabilities as soon as they are introduced into the software makes it extremely useful to prevent unnecessary costs of bug fixing at a later stage of the engineering process. It relates to our work, because the boot ROM code in embedded systems usually cannot be patched after construction and vulnerabilities can persist for years.

Gary Robinson Larry Conklin. OWASP Code Review Guide 2.0. Version RELEASE. OWASP. URL: `https://owasp.org/www-pdf-archive/OWASP_Code_Review_Guide_v2.pdf` **(visited on 04/20/2020)**
The "OWASP Code Review Guide" is an in-depth treatise on code review born from the OWASP Testing Guide. It discusses many aspects of code review and goes much further than will be necessary in our work. While it focuses on web security, almost all the principles of code review and code analysis apply to any kind of software.

3.2 Power Analysis

Side-channel attacks have been known for decades but with the emergence of the internet of things they are more relevant than ever. Much of the scientific work on this topic is on smart cards and how they were vulnerable to this type of attack at the end of the last millennium. Due to the nature of the ChipWhisperer tool we used for this project we were able to focus on differential power analysis.

Jun B. Kocher P. Jaffe J. "Differential Power Analysis". In: CRYPTO 1999: Advances in Cryptology (1999), pp. 388–397
This paper discusses different methods for power analysis and examines their effectiveness in extracting secret information from tamper resistant devices. It also mentions methods of making crypto systems that can operate securely in a system that leaks information, which is outside the scope of our project.

P. Socha, J. Brejník, and M. Bartik. "Attacking AES implementations using correlation power analysis on ZYBO Zynq-7000 SoC board". In: 2018 7th Mediterranean Conference on Embedded Computing (MECO). 2018, pp. 1–4
In June 2018 Petr Socha, Jan Brejník and Matěj Bartík released a paper that details a power analysis attack on AES using the exact microchip we used for our testing. Instead of designing a custom board they used a modified ZYBO board. Their work details the process of attacking both software and hardware implementations of AES-128. It served as proof that such an attack was feasible for our target.

3.3 Glitching

When glitching an embedded device one can target the device clock, or one can induce the glitch on the power supply line. The tutorials for the ChipWhisperer tool provided by NewAE Technology were used as a starting point and adapted to our target.

Colin O'Flynn. Fault Injection using Crowbars on Embedded Systems. 2016
One of the most important motivations for this project was the ChipWhisperer tool developed by Colin O'Flynn. His papers on fault injection were the basis for the glitching attacks performed during this project. His text on using crowbars discusses many different fault injection methods and gives a good overview of the most important ones.

Francesco Palmarini Claudio Bozzato Riccardo Focardi. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. 2019
This paper talks about voltage glitching and introduces a new technique, comparing it to popular existing voltage glitching approaches. It is especially relevant for our work, because the authors give an in-depth analysis on the construction of glitch pulses. They report on six unpublished vulnerabilities in micro controllers from three different manufacturers. All the attacks can be performed in a black-box setting and are nondestructive, which is especially useful because it corresponds to our testing setup.

3.4 Fuzzing

Fuzzing is a widely accepted and well researched technique that complements traditional software review and testing. It plays a major role in the development process for traditional software systems but is less frequently used for embedded systems.

Andreas Reiter. In-Memory Fuzzing on Embedded Systems. Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, 2012
The thesis by Andreas Reiter covers most of the important aspects of fuzz testing. It gives a comprehensive overview of testing methods and their justifications. It discusses the most important bugs that can be found with fuzz testing and how they are exploitable. While the main topic of in-memory fuzzing is not directly applicable to our project setup, the methodology of developing a fuzzer provided valuable insights into the technique.

4 Approach

4.1 Goals

> *If you don't know where you are going, any road will take you there.*
>
> – Lewis Carroll, *Alice in Wonderland*

It is important to define goals at the beginning of security research, and to think carefully about what you are trying to achieve. It is easy to get lost in just trying one attack scenario after another if there is no plan to follow. In this case, we set ourselves the same goals that a potential attacker would have in breaking the security of an embedded secure boot implementation. Objectives and desired gains for an attacker in doing so are:

- Breaking encrypted boot images: Some processors feature internal crypto engines (e.g. AES-256) to load boot images that have been encrypted to ensure confidentiality of the software and protect the intellectual property of the designer. Breaking the encryption by, for example, extracting the secret decryption key that is stored on the chip would allow an attacker to reverse the software and steal intellectual property.

- Bypassing boot image authentication: Some processors feature internal authentication mechanisms (e.g. RSA-2048) to allow only boot images with a correct signature, to ensure that only authorized software can be run. Bypassing the authentication would allow an attacker to run arbitrary code on the processor and to place backdoors in the system or replace the entire boot image with a malicious one.

- Extracting confidential boot ROM code: Some processors contain an internal read-only memory (ROM) section where they store critical proprietary boot code that also serves as the ROT when secure boot is used. Because of its sensitive nature, this code is usually not available to the public and is protected against extraction. Getting access to this code by, for example, overriding the read-out protection would grant an attacker a considerable advantage in finding unpatchable vulnerabilities in the boot ROM that could potentially be exploited.

4.2 Scope

For this thesis, the focus is on overriding security mechanism provided by the chip manufacturer. Attacks that require physical access are particularly in-scope.

4.3 Means

To achieve these goals we will use means like fuzzing and code analysis, especially of critical sections like the FAT driver. Even though the authentication may be first step in checking a boot image and might reject illicit images, it still needs to be loaded by the FAT driver first. Power analysis has already been done by several other researchers for this chip and will only be visited briefly. Electromagnetic fault injection (EMFI), although interesting, will not be a topic for this thesis as in requires additional and different kinds of hardware tools. Instead, the other fault injection attacks like voltage or clock glitching will be pursued.

5 System Design

This chapter will describe the process from defining the system requirements for performing the planned experiments to the implementation in hardware. First, the system and hardware requirements to perform the experiments were defined.

5.1 Requirements

5.1.1 Power Analysis Requirements

Power analysis requires the capability to measure and record minute changes in the power consumption of the target. The sample rate of the ADC (analog-to-digital converter) capturing this signal must satisfy the Nyquist sampling criterion. Often, the signal needs to be amplified (ideally directly on the target board) to maintain a high SNR (signal-to-noise ratio). The voltage regulators supplying power to the target must minimize output ripple and noise to provide a clean baseline. If the target has multiple supply rails, the measurement needs to be performed on the core supply rail, to capture the power consumption variations of the CPU itself. To correlate the power measurements to system events and determine what the CPU is doing at a certain point, a trigger system is required. In summary, the following requirements have been established:

- Measure and record power consumption of relevant target supply rails
- Sampling at higher than Nyquist rate of the target clock frequency
- On-board LNA (low-noise amplifier) for optimized SNR
- Filtered, low-noise power supply for clean measurement baseline
- Trigger system to correlate power analysis data to system events

5.1.2 Fault Injection Requirements

To perform voltage glitching, the CPU is deprived of power for a brief and well-controlled duration at a certain point in time. Usually this is achieved with a crowbar circuit; an electronic switch that short-circuits the target core power rail for the duration of the glitching pulse. For clock glitching, the pulse is instead injected into the system clock input of the target. In both cases, the timings of the pulses are the parameters that need to be determined for every new target and therefore need to be adjustable with high precision. Their stability over time is important to ensure repeatability of the glitching results. It is advisable to make the target core power supply voltage and system clock frequency adjustable. This offers the opportunity to operate the chip at the edge of its specified operating range, facilitating fault injection. This is

particularly useful for targets that are hard to glitch. A trigger system precisely controls the point in time when the glitch is injected, down to a specific instruction. Trigger sources are usually a combination of system events and precise time offsets. An example for a system event is a change in output state of a target pin to signal that it has arrived at a certain point in the program code execution. To protect sensitive support components like the LNAs from glitches and voltage spikes, a diode protection network is recommended. Glitching may in rare cases cause the target to suffer a latch-up event, an internal short-circuit condition in the integrated circuit of the target. If the power is not disconnected immediately, the high short-circuit current will likely destroy the target IC. To protect against this event, a high-speed electronic fuse is highly recommended. It also protects against errors in the control of the voltage glitching crowbar circuit, accidentally shorting the supply for too long. In summary, the following requirements have been established:

- Inject adjustable length voltage glitches into the target power supply with precise timing
- Inject adjustable length clock glitches into the target clock input with precise timing
- Variable target core power supply voltage
- Variable target system clock frequency
- Precise timing trigger system to target specific CPU instructions and ensure repeatability
- Protection network to protect sensitive support components from voltage spikes
- Fast overcurrent protection in case of target latch-up or control error

5.1.3 Fuzzing Requirements

To succeed with a fuzzing test, usually many fuzzing runs are required. It is important to minimize the time required by a single run, maximizing the number of runs that can be performed in a certain time frame. Automation is required to constantly generate new inputs to the system being fuzzed and evaluate the response. Boot image fuzzing adds a complication in that the image generation and fuzzing control cannot usually run on the target device, because it needs to reset between runs. An external test server is therefore used to generate the manipulated boot images for fuzzing the target. First, the test server loads a new boot image into the targets boot image storage. This is achieved by either switching the storage between the test server and the target, or by the test server taking control over the target and commanding its CPU to write the image into the storage. The target will then reset, load and run that image. The test server monitors the target and logs if the result is a success or a crash. This concludes one fuzzing run and the test server will proceed with the next run, using a different boot image. In summary, the following requirements have been established:

- Creation and manipulation of target boot images on an external test server
- Switching of boot image storage between target and external test server
- Control (start, stop, reset) of the fuzzing run by the external test server
- Evaluation of the fuzzing run result by the external test server

5.2 Generalized System Design

A generalized system design was elaborated that would satisfy the requirements above and could be adapted and implemented to different target devices. The block design of this system is shown in figure 5.1. Many of the requirements are unique to this kind of experiments and the necessary electronics are usually not included in off-the-shelf development boards. Modifying them may be possible, but entails several disadvantages and difficulties:

- Power traces on the PCB need to be cut to allow for power measurement
- If the power is supplied to the target over internal power planes, cuts are very hard to do
- Small SMD decoupling capacitors need to be removed for effective voltage glitching
- Additional parts need to be soldered to the PCB (e.g. glitching transistors, shunt resistors)
- Available connectors and interfaces may not match the ones on the external tools
- Usually not prepared for automated testing
- Peripherals like ethernet or display interfaces are not needed and increase the noise floor
- The PCB modifications and countless wires may reduce reliability and repeatability

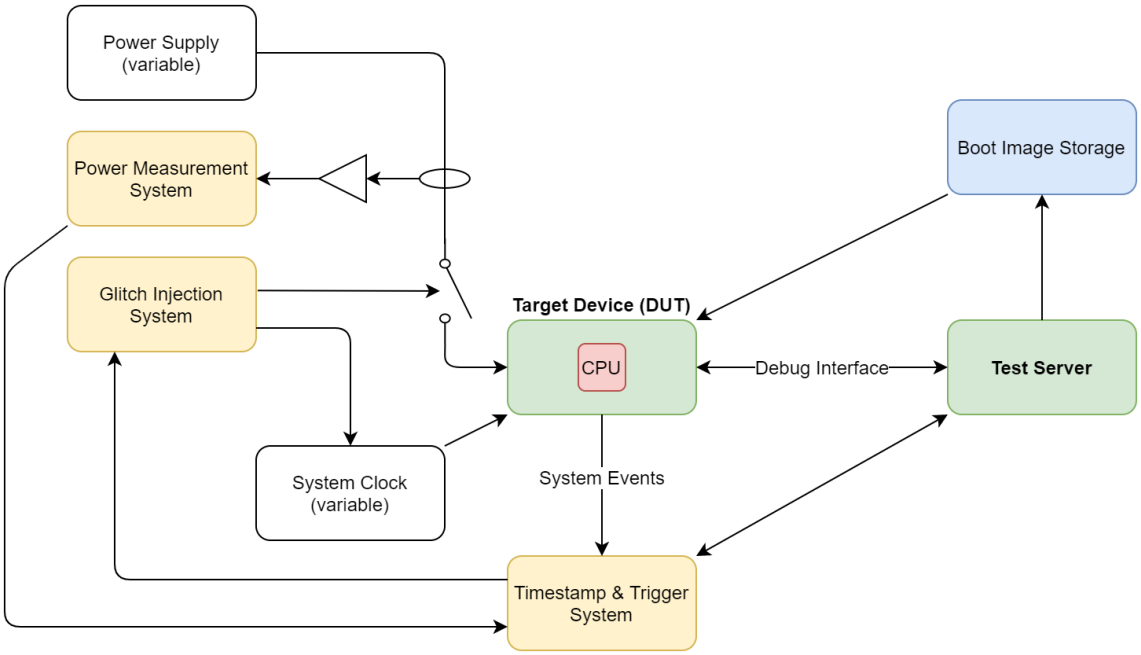


Figure 5.1: Generalized System Block Design

A successful example of a target board that was modified to support power analysis and glitching experiments can be seen in figure 5.2 to visualize what the challenges involved with this method are. Based on the points listed above, the decision was made to develop a custom board.
The most difficult and labor-intensive part of developing the system depicted in figure 5.1 are the power measurement, glitch injection and trigger systems, marked in yellow. They involve high-speed analog and digital design, FPGA development and then firmware and software

development to control the tools. Luckily, a ready-made tool for experimenters and researchers of hardware security that covers all this does already exist; the ChipWhisperer. For our requirements and budget, the ChipWhisperer-Lite was deemed the best option. The Nano does lack some features necessary for this project, and the Pro did not fit our budget.
NewAE Technology, the manufacturer of the ChipWhisperer, does also sell a carrier board called the "CW308 UFO" to interface with target boards for different microchips. This would normally be the quickest way to test a new processor with the ChipWhisperer. But it does also limit the designer of a new target board to the available interfaces, voltages and PCB design. For this project, which also aims to support more complex system-on-chip targets, the CW308 UFO carrier was deemed too restrictive. This is probably also the reason why NewAE Technology developed their CW305 FPGA target board separately. It features a Xilinx Artix-7 FPGA which includes the same programmable logic as the ZYNQ-7000 SoC used for this thesis, but without the ARM processors, the authentication features or the boot ROM. For these reasons, we decided to develop a custom board without relying on the CW308 UFO carrier.

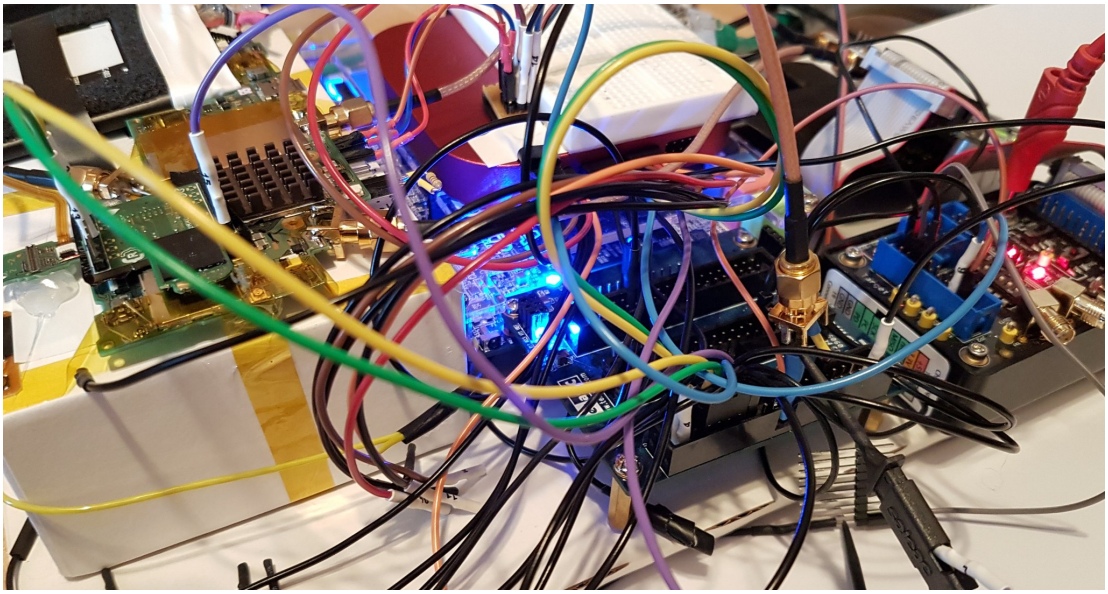


Figure 5.2: Example of Experimentation Setup With Modified Target Board
Source: Bernhard Froemel. January 29, 2018. Twitter @bernfroe

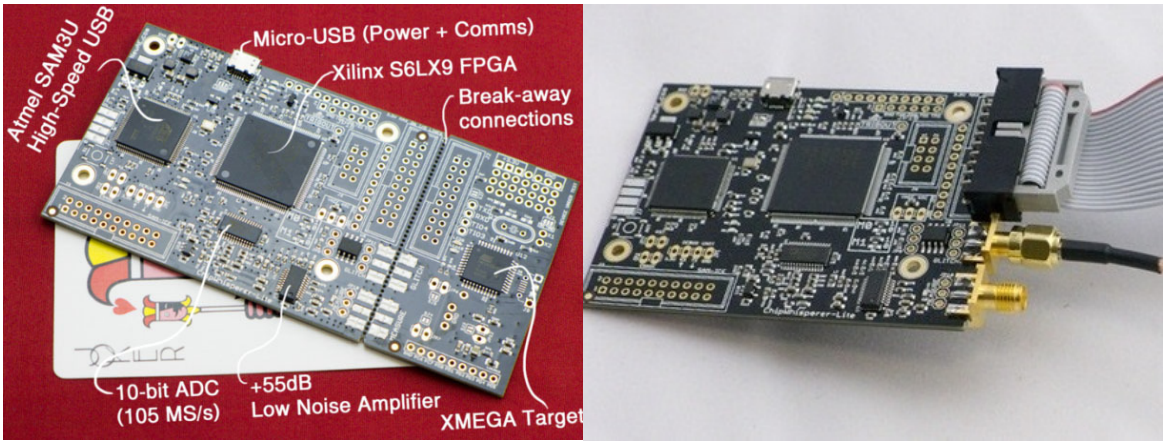


Figure 5.3: ChipWhisperer-Lite
Source: NewAE Technology, newae.com

5.3 ZYNQ-7000 SoC System Design

After defining the requirements and elaborating a generalized system design, an implementation specifically for the ZYNQ-7000 SoC that was to be used as an example for this thesis had to be done. First it was determined, what peripherals would be necessary to support the design:

- SPI flash memory for boot image storage
- SD card interface and multiplexer for boot image storage (interesting for fuzzing)
- JTAG header for programming and debugging
- Adjustable voltage power supply
- Adjustable frequency system clock generator
- Interface to the ChipWhisperer-Lite (signals, triggers, serial port)
- Test server and with communication interface (debug interface, serial port)

Many other peripherals that are commonly found on development boards were not included, including USB, Ethernet, audio oder video codecs or displays. The DDR memory was also omitted, because the software for the experiments is small enough to run from the internal on-chip memory (OCM), and because it would have complicated the PCB layout quite a bit. The disadvantage is that Linux could not run on this board with that little available memory, but this was a trade-off we were willing to make. If the AES encryption key is stored in the internal BBRAM, a backup battery is required to supply the memory when the board is powered off. However it only takes a couple of seconds to reprogram the keys over JTAG, so the backup battery was also omitted.
The ZYNQ-7000 SoC family consists of different devices, some feature a dual-core CPU, others only a single one. Also there are different sizes of the integrated programmable logic available, with the bigger devices getting quite costly. To stay withing budget, we selected the smallest device possible; the XC7Z007S with a single ARM CPU and the smallest programmable logic array. It fits our requirements because we don not need the logic array and the boot code runs on a single core only anyway. The smallest chip package (CLG225 with 225 pins) could however not be used, because the pins to boot from the SD card are not available, which we require for the experiments. That is why the next larger package (CLG400 with 400 pins) was selected, even though the higher number of I/O pins was not necessary.
To be able to load a boot image from the test server to the SD card, and then boot from the same card with the ZYNQ, there were three possible solutions: The first idea was to use a WiFi-enabled SD card and load the boot image from the test server over WiFi. But it was unclear how reliable these cards are and if they would work with the ZYNQ's boot ROM code. Another possibility would be to emulate and SD card, but this idea was quickly abandoned after realizing how much effort that would require. The simplest solution, which was ultimately chosen, was to switch the card between the test server and the ZYNQ with a multiplexer IC and powering it off in between. This is functionally and electrically equivalent to changing the card between two different sockets by hand and was therefore guaranteed to work.
A Raspberry Pi 3B was selected for the test server, because with its small size it can fit directly on the experimentation board, features a UART serial interface and GPIO pins to connect to the ZYNQ's debugging signals. Also it is powerful enough to run the test scripts, but can easily be left on for several days when performing longer experiments.

5.4 ZYNQ-7000 SoC Hardware and Board Design

In a word play on "breakout board", which is a PCB that "breaks out" the pins of a microchip to connectors, the board that was designed for experimenting with physical attacks in this thesis is called the "ZYNQ Break-In Board" (ZBIB).
To allow for rapid firmware development, the pinout of the peripherals was kept mostly consistent with the one on the Digilent ZedBoard (Rev. D2), which also features the same ZYNQ-7000 SoC. Most of the peripherals like the Ethernet interface, the DDR memory and the HDMI video output are not present on the ZBIB, because they are not necessary for our application and increase cost, complexity and the noise floor of the measurements. Pinout compatibility with the Zedboard allows to use its templates in Vivado for configuration of the chip and drivers, with only minor modifications. This saves the effort of writing a custom board file.

5.4.1 Schematics Design

This section explains important design decisions and parts of the ZBIB circuit. The full circuit diagrams can be found in the appendix.

Power Supply Circuit

The power supply performance is critical to the success of the power analysis and glitching experiments. A high noise floor will impair the measurements and an unstable supply might collapse under the pulse load of voltage glitching. It is also a part of the design that needs special consideration when adapting to a different target chip, because power requirements can vary. First, assess what supply rails are needed and what their current requirements are. Sometimes there may be supply rails with the same or overlapping voltage specifications that could potentially be combined. If analog and digital supply rails are combined, switching noise may be introduced from the digital into the analog domain. In this case, a low-pass filter in series with the analog supply is recommended to reduce this noise. Next, consult the datasheet for the typical and maximum supply current of each power domain and select the voltage regulators accordingly.
For the ZBIB with the ZYNQ-7000 SoC, the following power supplies are required:

- +3.3V: Supply for the ZYNQ I/O system and all peripherals
- +1.8V: Supply for the ZYNQ auxiliary and PLL rails
- +1.0V: Supply for the PS (processor) and PL (logic) rails

When selecting the voltage regulators, a choice between linear (LDO) and switching regulators (DC/DC) needs to be made. LDOs have superior noise rejection (PSRR) and provide a cleaner output voltage, but the voltage difference between the input and output needs to be small because the excess is converted to heat. DC/DC converters are more efficient in this regard, but the output voltage may exhibit ripple voltage from the internal switching and is not as smooth.
For the ZBIB, a hybrid approach was chosen to combine the advantages of both regulator

types. The +3.3V and +1.8V rails are not measured and therefore less sensitive to noise (apart from the PLL supply, which is low-pass filtered). They will be supplied by switching regulators. The two +1.0V core supply rails are sensitive to noise because they need to be measured for the power analysis experiments. Two DC/DC converters will bring down the voltage to +1.2V, where one LDO each will take over and regulate the voltage to a nominal +1.0V, smoothing the switching ripple and noise of the DC/DC converters. Additionally, the exact output voltage of the LDOs may separately be adjusted between +0.893V and +1.1V with a rotary switch. Operating the core supply rails at the edge of their operating range may help introducing errors when performing glitching attacks. The adjustment circuit was specifically designed that the output voltage will not exceed the maximum supply voltage of the ZYNQ, if the the switch fails because of dust or mechanical deterioration.
For the switching regulators, two ISL8203 parts were selected. They combine two independent voltage regulators into one package, providing a total of four supply rails. They also allow for a fast and easy design, because the switching transistors and inductors are already included into the package. This reduces the time to design the circuit and the risk of a suboptimal layout when routing the board. If their switching ripple would still impair the measurements despite the LDOs, they feature an optional synchronization input that could be synchronized to the ADC sample rate to effectively cancel their influence on the ADC measurements. For the board bring-up it is useful to first power up only the voltage regulators and check if they are supplying the correct voltage to avoid the risk of destroying other components. Then, the jumpers can be soldered closed to connect the rest of the system once the correct operation of the power supply is confirmed. The resulting circuit is shown in figure 5.4.

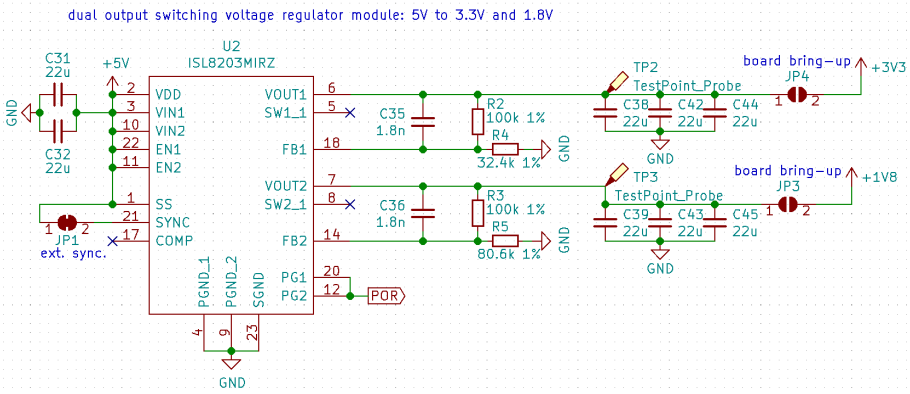


Figure 5.4: ZYNQ Break-In Board: One of Two DC/DC Voltage Regulator Circuits

For the LDOs, a choice had to be made between the TPS7A89 and the NCP59744. The former has the advantage of providing a dual output, like the ISL8203, so only one of them would be needed. But the NCP59744 has better load regulation and noise rejection (PSRR), lower droput voltage and also a higher current capability, so two of them were selected. For most chips with multiple power domains, the power-up and power-down sequencing of the supply rails needs to be considered. Sequencing in the ZYNQ does not need to be controlled, as long as all supply voltages are in the valid operating range when the power-on reset (POR) is released. To ensure this, the power good (PG) pins of the regulators are connected to the POR input of the ZYNQ. The resulting circuit is shown in figure 5.5

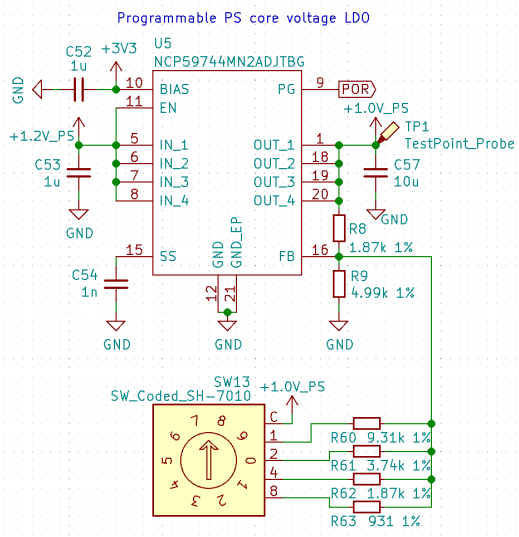


Figure 5.5: ZYNQ Break-In Board: One of Two Linear Voltage Regulator Circuits

The power input of the ZBIB is protected against overvoltage spikes and overcurrent with an LTC4361 circuit. The overcurrent protection is particularly important, because it protects the components against accidental short-circuits. The crowbar circuit on the ChipWhisperer consist of an electronic switch that shorts the target core power rail to introduce glitches. If the switch accidentally stays closed for too long, excessive current will flow and destroy the switch or the power supply. In rare cases, glitching the target may cause it to suffer a latch-up event, an internal short-circuit condition in its integrated circuit. If power is not disconnected immediately, the high short-circuit current may destroy the target chip. In such an event, the overcurrent protection circuit immediately removes power to the rest of the ZBIB faster than a conventional fuse. To reset it, the error that caused it needs to be resolved and the power cycle on the board triggered. The protection circuit is shown in figure 5.6.

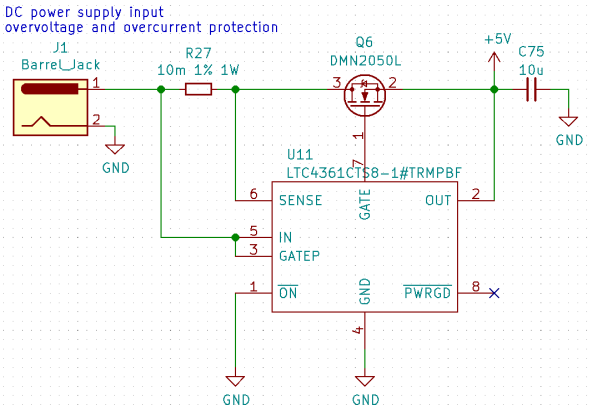


Figure 5.6: ZYNQ Break-In Board: Power Input and Protection Circuit

Power Analysis and Glitching Circuit

To minimize risk and speed up the design, this part of the circuit design was taken from the NewAE Technology CW305 FPGA target board. It consists of a low-pass filter to remove as

much noise as possible from the power measurements, a current shunt to measure the power consumption, a low-noise amplifier (LNA) to amplify the signal, and SMA coaxial connectors to connect to the ChipWhisperer or an oscilloscope. There are two of these circuits, one for each rail. One SMA connector provides the unamplified current shunt signal, intended to be connected to the measure port of the ChipWhisperer, which already has an internal LNA. Another SMA connector outputs the amplified current shunt signal and is intended to connect to an oscilloscope for additional debugging. The last SMA connects to the glitching port of the ChipWhisperer and its crowbar circuit, to inject voltage glitches. Diodes protect the LNA and other parts of the circuit from inductive spikes caused by the glitching pulses. The circuit is shown in figure 5.7.

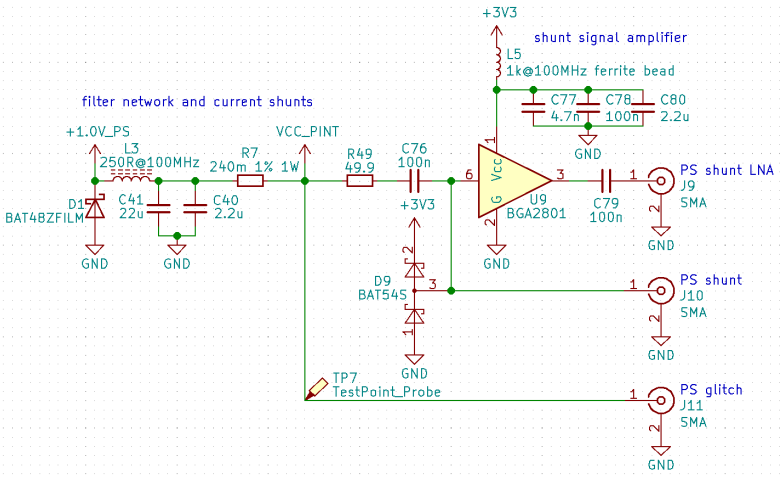


Figure 5.7: ZYNQ Break-In Board: One of Two Low-Noise Amplifier Circuits

Clock Switching Circuit

The ZBIB features an onboard oscillator to provide a stable system clock to the ZYNQ. It passes through an SI53361 clock multiplexer with output buffers. The second clock input to the multiplexer is the clock output from the ChipWhisperer, which can inject glitches into the clock signal. With a switch, the clock output can be selected from either input. The SI53361 buffers the clock signal and outputs it to the system clock input of the ZYNQ and also back to the ChipWhisperer for reference. The resulting circuit is shown in figure 5.8.

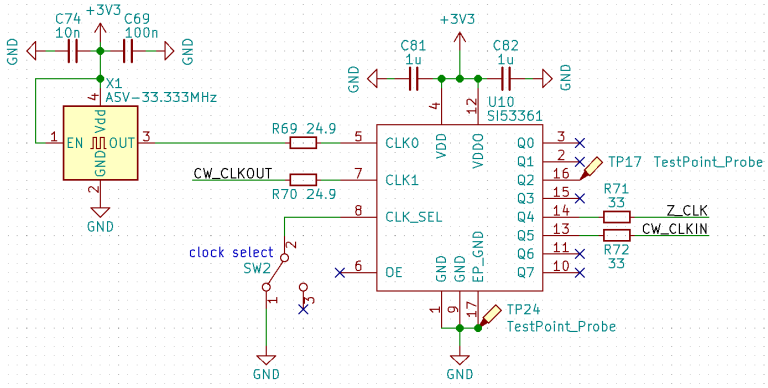


Figure 5.8: ZYNQ Break-In Board: System Clock Circuit

SD Card Switching Circuit

The ZYNQ-7000 SoC on the ZBIB can boot from one of two different sources; the SPI flash memory or the SD card. The latter is particularly interesting for fuzzing, because the interface is more complex and therefore the chance of finding vulnerabilities is higher. To automate the process of loading a new boot image on the SD card and then booting from it with the ZYNQ, an SD card switching circuit was designed for the ZBIB. The TS3A27518 multiplexer was selected to switch the interface of the SD card between that of the ZYNQ and that of the Raspberry Pi used as the automation test server. A MIC2025 high-side switch can cut the power to the SD card to reset it, because switching between two different hosts without powering off the card might lead to problems if the card is in an unexpected operating state. The resulting circuit is shown in figure 5.9.

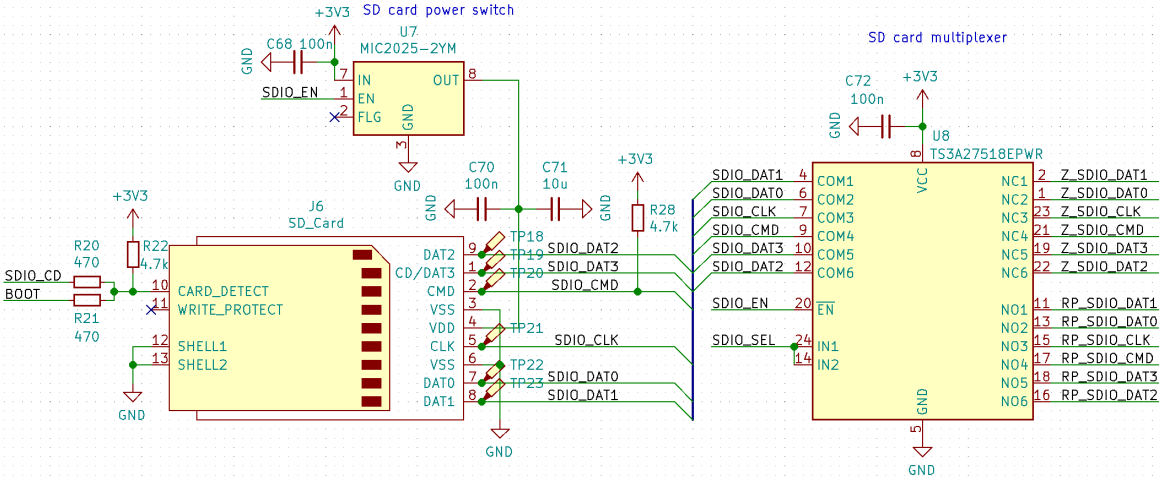


Figure 5.9: ZYNQ Break-In Board: SD-Card Circuit

5.4.2 Board Layout

The board was designed as a six-layer PCB with one ground, two power and three signal layers as a compromise between cost and signal integrity. Some parts of the boards require fairly tight tolerances, mainly around the vias in between the 0.8mm pitch balls of the ZYNQ's BGA package. This further narrowed down the number of PCB manufacturers, which was already limited because of the COVID-19 pandemic in China at the time. The layout was done with the KiCad CAD package and the decision was made to release the design files as open source hardware to give back to the community. Some of the component footprints were available in the KiCad integrated footprint library, other were downloaded from SnapEDA or component search engine by SamacSys. On components that may need to be removed or re-soldered, like the ZYNQ bypass capacitors, the slightly larger footprints for hand soldering have been used. The ZBIB design includes some fairly high-speed buses like the SD card SDIO interface, quad-SPI bus of the flash memory and JTAG interface. The traces of these nets have been length-matched to avoid skewing the signals because of different propagation delays. The log of the design rule check (DRC), which was set up in the beginning with the specifications from the PCB manufacturer, was checked to make sure that no violations were left before sending the design off to manufacturing. The final layout is shown in 5.10, images of the individual layers can be found in the appendix.

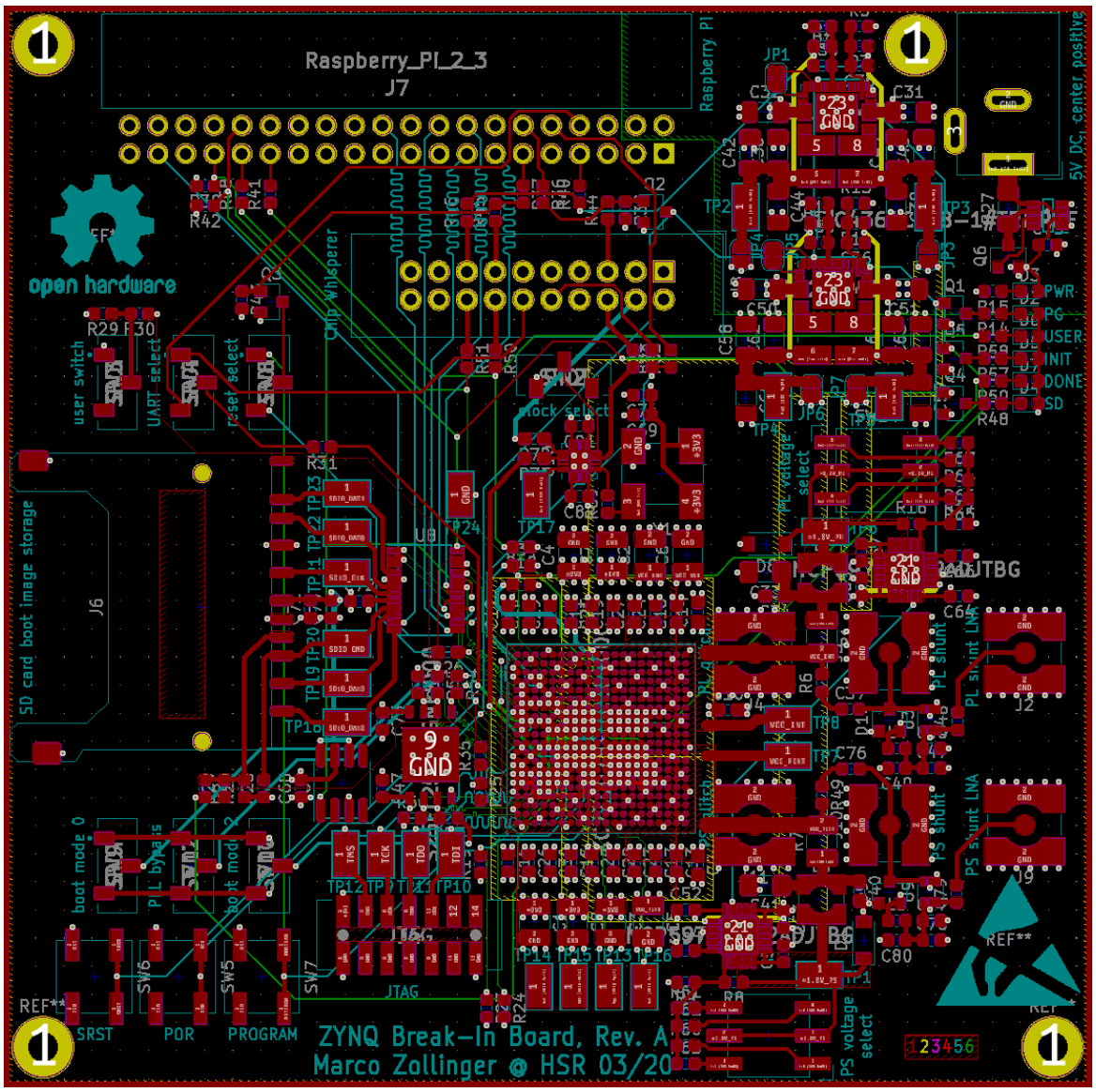


Figure 5.10: ZYNQ Break-In Board: Finished Layout, Overview

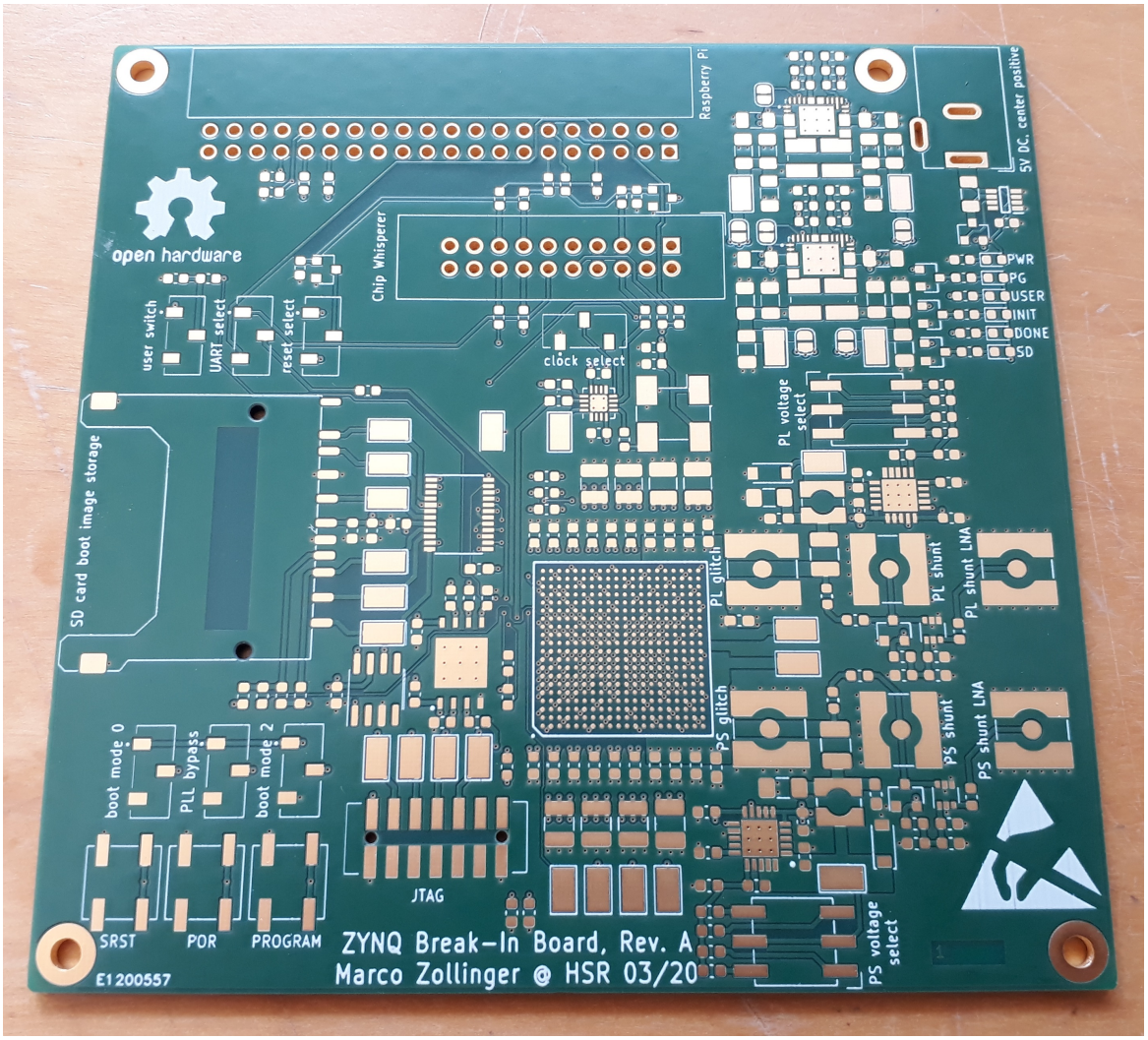


Figure 5.11: ZYNQ Break-In Board: PCB From Manufacturer

5.4.3 Board Manufacturing and Brief Function Test

The original plan was to have the PCB assembled in China, but the risk of delays was considered too high, and assembly in Europe did not fit the budget. Therefore it was decided to assemble the PCB at home. In a professional assembly process, the solder paste is applied on the PCB using a stencil, the parts are placed with a pick-and-place machine and then the board is soldered in a temperature-controlled reflow oven, closely following the temperature profile. We did order a stencil with the PCB and the parts can be placed with tweezers and a steady hand (and much patience), but a reflow oven was not available. However, some hobbyists are using modified toaster ovens to reflow their PCBs, and that is what we did as well. The oven was lined with aluminum foil to improve thermal insulation. This is important, because if the oven takes too much time to reach the temperature where the solder melts, then sensitive components might be damaged or plastic parts like connectors could melt. The toaster oven does not have a useful temperature controller, so the temperature measured (with a thermocouple) needs to be compared to the reflow profile and the oven adjusted by hand accordingly. A few test runs have been conducted to characterize the oven and the rate of the temperature rise. The resulting graph and the modified oven can be seen in figure 5.12.

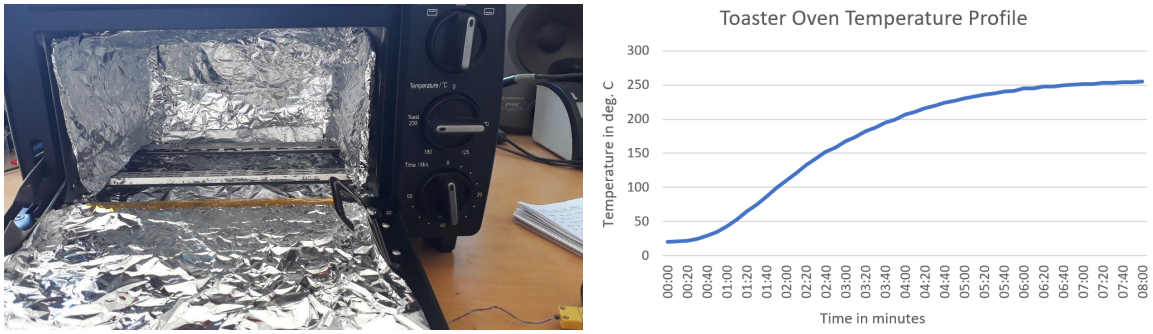


Figure 5.12: Modified Toaster Oven And Temperature Ramp-Up

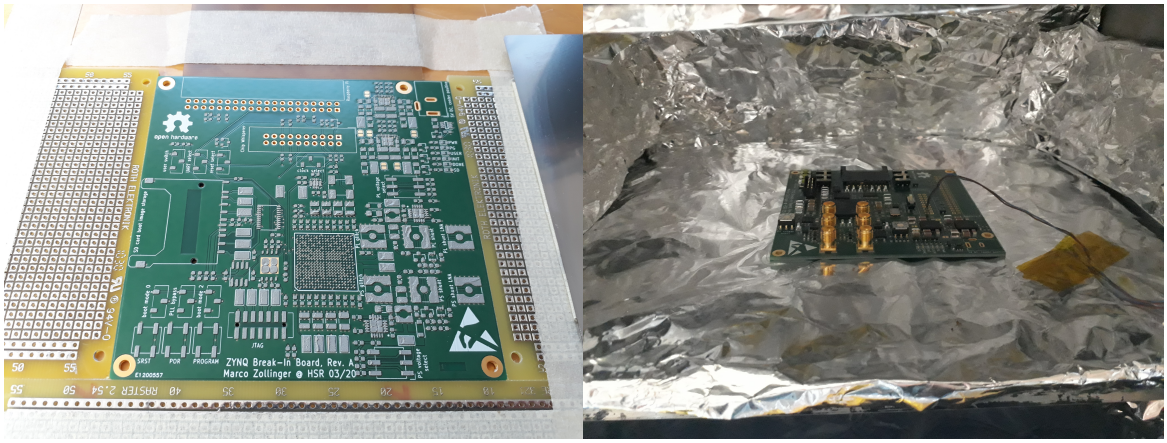


Figure 5.13: Board After Paste Application and Before Reflow

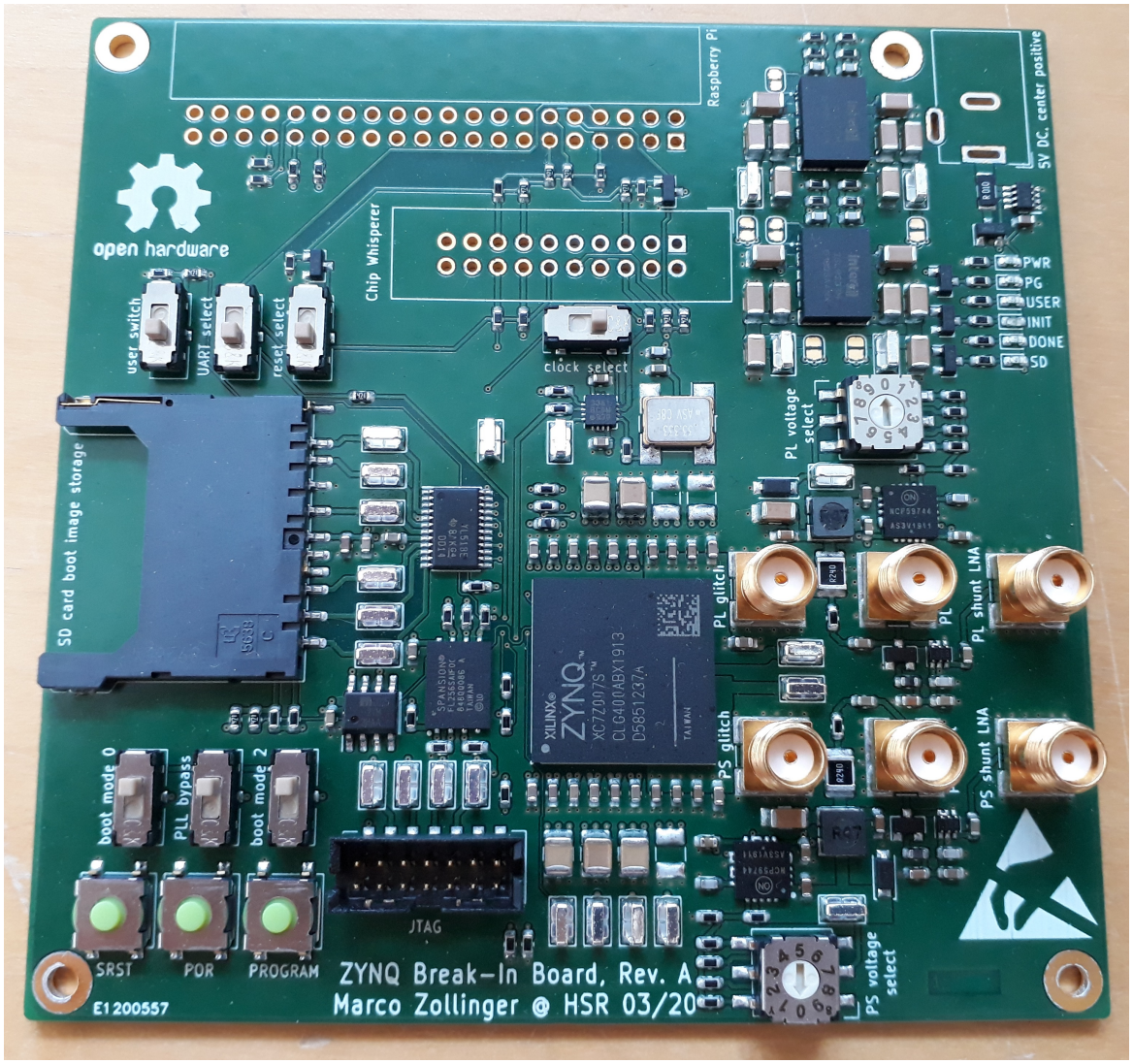


Figure 5.14: ZYNQ Break-In Board After Successful Reflow

After the successful reflow of the board, a brief function test has been conducted. First, a visual inspection confirmed that all components have correctly been soldered in place. No short circuits were detected on the power rails. The voltage regulators were powered up with the rest of the circuit disconnected and the proper operating voltage was verified. Then, the bring-up jumpers were close, connecting the circuit to the power supply and a simple application to blink an LED with the ZYNQ was uploaded over JTAG. This basic function test does not cover the operation of all of the peripherals, but at least confirmed that the ZYNQ and its support components were working fine.

5.4.4 Errata Rev. A

Even though the ZBIB design was done carefully, a few errors have managed to sneak into this first revision:

- Transmit (TXD) and receive (RXD) pins of the ZYNQ UART are interchanged. The issue has been fixed by cutting the traces and crossing them with enameled copper wire.

- The USER LED is connected to the PS_MIO_VREF pin instead of a MIO pin. The issue has been fixed by cutting the trace and connecting the LED to a MIO pin.
- It seems like the switching regulators are interfering with the WiFi connection of the Raspberry Pi. The workaround is to use wired networks (Ethernet LAN) on the Pi instead.
- Some labels for the LEDs are in the wrong order. A piece of paper with the correct order has been put in place.
- Test points and jumpers are not labeled because of an error in exporting the board manufacturing data. The locations can be looked up in the documentation.

With all known errors resolved the board was fully operational for the purpose of this project.

6 Code Analysis

To thoroughly test an embedded system, one must look at the software code itself. Code analysis is an important tool in software engineering. It allows the automation of time-consuming processes to ensure code quality. Dedicated software tools and plugins can detect certain vulnerabilities and bad code structures at the time they are written. The use of such tools is especially important because adversaries looking to attack or exploit a system will likely deploy these same tools to look for weaknesses. [6]

6.1 Manual Code Analysis

The most basic way to analyze code is by hand. To find potential weaknesses or attack vectors we look at any documentation of the code itself or patches and updates. Bug fixes are especially interesting from an attacker's point of view because they reveal weaknesses in older versions of the code. Especially in embedded systems with no network connection one can often find outdated software versions.
The boot ROM code is seldom open source, because if a vulnerability is found in the code it is impossible to patch and whole chip series would need to be replaced or secured in a different way. In our work we looked at code of the FSBL example provided by Xilinx for the ZYNQ-7000 SoC. We suspected code snippets could have been reused from the boot ROM code and a vulnerability in the FSBL could also indicate the existence of the same vulnerability in the boot ROM. Third party libraries also present a risk for the companies using them because they have less control over the development process.

6.1.1 Methods to Look out For

The most common vulnerabilities in C are related to buffer overflows and string manipulation. Most of the time this would lead to segmentation faults, but through specially crafted input values, adapted to the architecture and system environment, one could cause arbitrary code execution. [22]

Some of the methods to look for during an analysis are:

```
gets()
strcpy()
strcat()
strcmp()
sprintf()
snprintf()
```

These methods do not check for buffer length and are vulnerable to buffer overflows. The strcpy method for example may overwrite memory regions adjacent to the intended destination, while printf could potentially leak secret information. [22]

The following code snipped shows a potential vulnerability when using the strcpy method:

```
char str1[10];
char str2[]="abcdefghijklmn";
strcpy(str1,str2)
```

Another type of vulnerability is concerned with string formatting attacks. The functions above all take a "format string" as an argument. The mitigation of this type of vulnerability is to not let the user control the format string with user input. [22]

```
printf()
fprintf()
sprintf()
snprintf()
```

6.1.2 Manual Code Analysis for the Xilinx FSBL

The goal of the manual code analysis was to get an idea of how an attacker would approach the topic, with the goal of finding exploitable vulnerabilities in open source code. This allowed us to formalize the method to a certain degree and generalize it for a manufacturer's use during the engineering process.

Code analysis obviously requires access to the source code. From an attacker's point of view this is not always possible. For our project we did not have access to the boot ROM code of the ZYNQ-7000, therefore we took a closer look at the example FSBL provided by Xilinx first. To expedite the search, we copied the source code into a text editor and performed string searches on the vulnerable methods mentioned above.

Special emphasis was put on the FAT file system library used. The implementation used by Xilinx was developed by ©ChaN, 2018. The Figure below shows the security patches since the version used by Xilinx.

```
R0.14 (October 14, 2019)
  Added support for 64-bit LBA and GUID partition table (FF_LBA64 = 1)
  Changed some API functions, f_mkfs() and f_fdisk().
  Fixed f_open() function cannot find the file with file name in length of FF_MAX_LFN characters.
  Fixed f_readdir() function cannot retrieve long file names in length of FF_MAX_LFN - 1 characters.
  Fixed f_readdir() function returns file names with wrong case conversion. (appeared at R0.12)
  Fixed f_mkfs() function can fail to create exFAT volume in the second partition. (appeared at R0.12)

R0.13c (October 14, 2018)
  Supported stdint.h for C99 and later. (integer.h was included in ff.h)
  Fixed reading a directory gets infinite loop when the last directory entry is not empty. (appeared at R0.12)
  Fixed creating a sub-directory in the fragmented sub-directory on the exFAT volume collapses FAT chain of the parent directory. (appeared at R0.12)
  Fixed f_getcwd() cause output buffer overrun when the buffer has a valid drive number. (appeared at R0.13b)
```

Figure 6.1: FAT File System Security Patches Since Xilinx's Version

For code used in the boot ROM, updates cannot be applied to an existing system and vulnerabilities would persist until the hardware is replaced. When searching for potential attack vectors, open source third party libraries provide an easily accessible surface.

For a manufacturer this means special care must be taken when selecting third party libraries to use within a system. Any code used this way must be reviewed carefully since, as seen in figure 6.1, bug fixes can reveal potential weaknesses of older software versions.

In our search we did not find exploitable weaknesses in the code, which indicates that Xilinx has reviewed their code thoroughly and has defined stringent coding guidelines.

6.2 Static Code Analysis

Static analysis is the examination of source code using a variety of methods such as data flow analysis. Static analysis tools can uncover issues such as memory leaks, buffer overflows, and even concurrency issues. Static analysis works by scanning one or more source files and creating a representation of the scanned source to analyze it. It can work on both the source code as well as the byte code. It can explore all code branches, even the ones not easily reached during normal code execution, which can be an advantage in comparison to dynamic code analysis.

6.2.1 CppCheck

CppCheck is a static code analysis tool for C/C++ that focuses on detecting undefined behavior and dangerous code constructs. We selected this tool because it is tailored to non-standard syntax common in embedded software and is frequently maintained by the developers.

The test result of zero false positives as well as zero false negatives strongly suggests that Xilinx may have used the same tool in their development pipeline and eliminated all warnings and errors prior to deployment. The only results were style warnings, which suggests differing style guidelines.

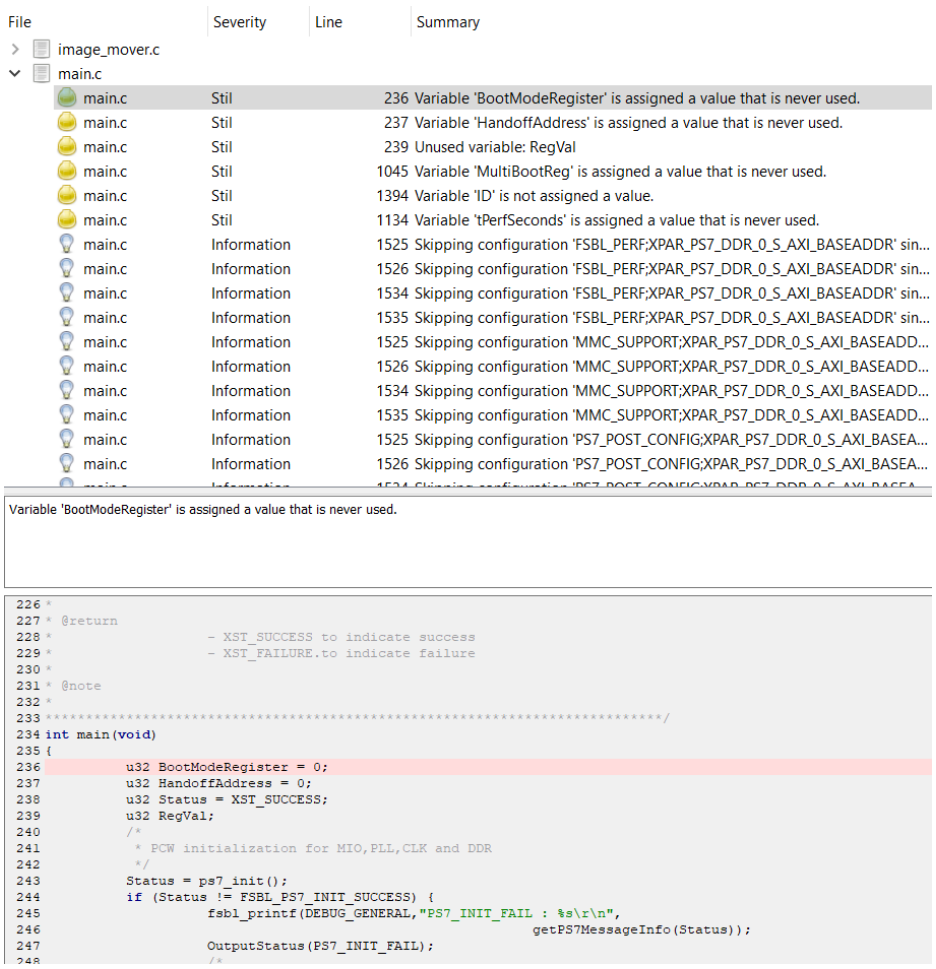


Figure 6.2: CppCheck Test Results of the Xilinx FSBL

7 Side-Channel Attacks

While many of the ciphers currently used for cryptography are considered mathematically secure, their implementations may be vulnerable to side-channel attacks. This type of attack exploits the fact that the intermediate values calculated in cryptographic operations correlate with the power consumption of the device as well as some other known information (e.g. plaintext or ciphertext, depending on the direction of the cryptographic operation). The big advantage of this type of attack is that it does not require a logical vulnerability. Even on flawless code these attacks can be performed. Typical targets of these attacks are not properly protected and expose encrypted secret information through side-channels.

7.1 ChipWhisperer

The ChipWhisperer is a tool developed by NewAE Technology for the purpose of automated power analysis as well as fault injection.

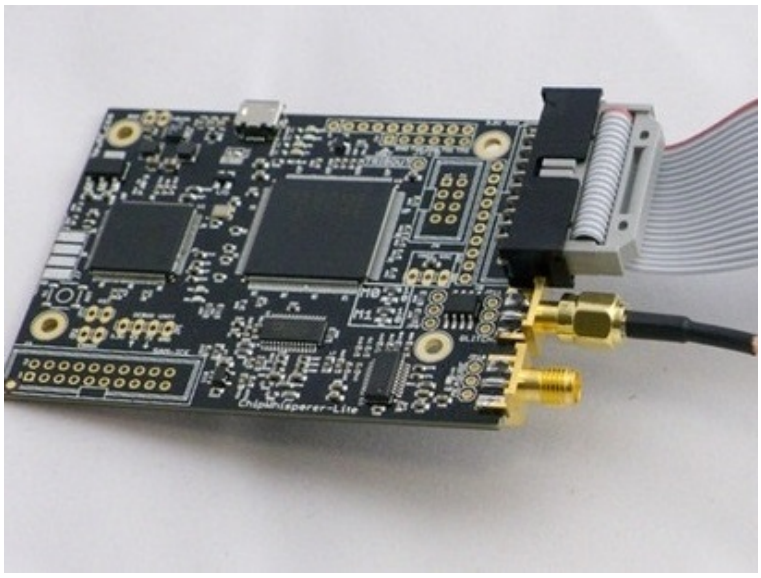


Figure 7.1: ChipWhisperer-Lite
Source: NewAE Technology, newae.com

The manufacturer provides a suite of tutorials as an introduction of working with the ChipWhisperer tool. For the purpose of this project we performed a power analysis attack on the target board included in the purchase. The board comes with an example firmware already configured for the tutorials, which includes encryption with AES-128.

7.2 Power Analysis

Power analysis is a form of a side-channel attack where an attacker measures the power consumption of a device. When used on a cryptographic system it can be used to non-invasively extract keys and other secret information. Simple power analysis (SPA) involves visually interpreting the power consumption of the system. This can be tedious and imprecise, which is where a tool such as the ChipWhisperer comes in handy. It allows the automated capture of a series of power traces of e.g. the secure boot process. A power trace is a set of power consumption measurements taken over time and across a cryptographic operation.

7.2.1 Differential Power Analysis

The power variations during the instruction sequences can be overshadowed by measurement errors and background noise. In such cases it is most often still possible to break the system using statistical functions tailored to the target algorithm. Differential power analysis (DPA) uses a large number of power traces that have been recorded while the device encrypts or decrypts different data blocks. They exploit the data dependency of the power consumption in cryptographic devices at a fixed moment of time as a function of the processed data. The mathematical principles behind this are discussed in Stefan Mangard. "Differential Power Analysis". In: *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Boston, MA: Springer US, 2007, pp. 119–165. ISBN: 978-0-387-38162-6.

Correlation Power Analysis (CPA) is a more sophisticated technique which we will not discuss here. It was introduced by [14] and [4]. An attack on the ZYNQ-7000 SoC using CPA was performed by Petr Socha, Jan Brejník and Matěj Bartik at the Czech Technical University in Prague, Faculty of Information Technology, proving the feasibility of such an attack. [20] For our project its inclusion would be a possible next step.

7.3 XMEGA Attack

The power analysis attacks on the XMEGA served the purpose of familiarization with the ChipWhisperer tool in addition to being an important method of attack. With encryption being the main method of securing the intellectual property represented by the software on embedded systems, a successful attack could potentially lead to large financial damage for the manufacturer. To perform the attack on the XMEGA target the corresponding tutorial for the ChipWhisperer tool served as a guideline. The attack was performed on the example implementation of the AES-128 algorithm. Capturing the power traces proved to be simple when using the premade scripts provided in the tutorial. Figure 7.2 shows an example power trace of 50 iterations.

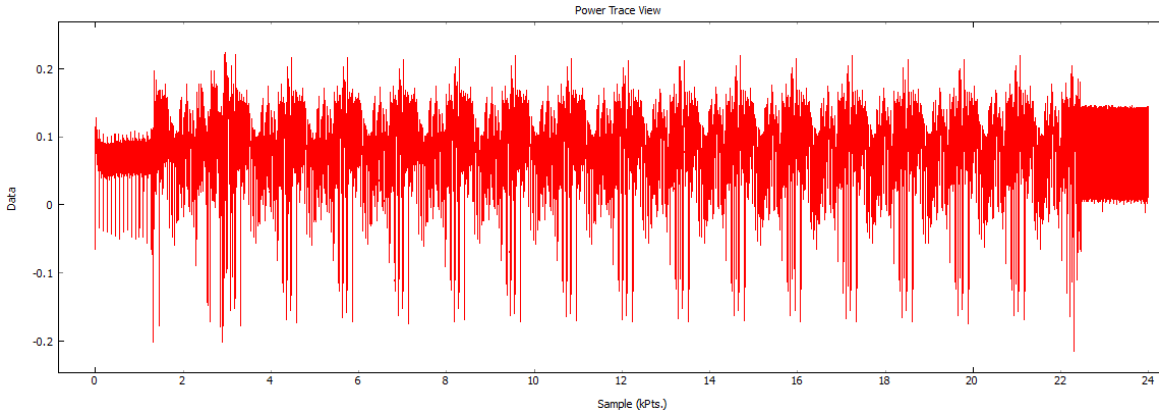


Figure 7.2: Power Trace

Analysis of the power traces was done using the attack script and was as trivial as running the Python attack script from the tutorial. The resulting secret key matched the key generated for the trace capture, indicating a successful attack.

The adaptation of this attack to our evaluation board was omitted due to time constraints and because such attacks have already been demonstrated for our target, but should not be too difficult.[20]

8 Glitching

Faults cause a piece of software to behave in an unexpected or unintended manner. For a system this could have severe consequences, and protecting systems from such faults is an important aspect of developing an embedded system. The term "fault injection" encompasses techniques that purposefully cause a fault to occur. It is a useful testing tool for systems constructed for high-radiation environments that can cause single-bit failures. [2] Previous work on fault injection has shown, that by carefully manipulating the faults introduced in a system it is possible to break cryptographic algorithms such as DES [9], AES [18] and RSA [1], that are believed to be secure mathematically.

8.1 Clock Glitching

Clock glitching is the method of inserting additional rising edges into the input clock of the device, with the goal of violating timing constraints in the target device. Most modern microprocessors have a pipeline and decode the next instruction while the current one is executing. Figure 8.1 illustrates this process.

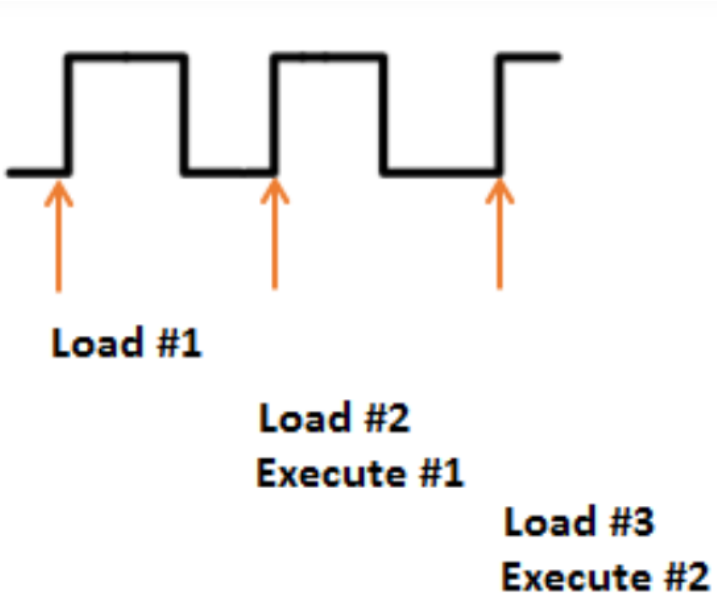


Figure 8.1: Device Clock and Instruction Pipeline

By introducing a glitch into the device clock we can create an additional rising flank. This can result in the clock triggers being too close together for the processor to have enough time to finish executing an instruction, effectively skipping it. Figure 8.2 illustrates this; instruction #1 does not have enough time to complete before instruction #2 replaces it.

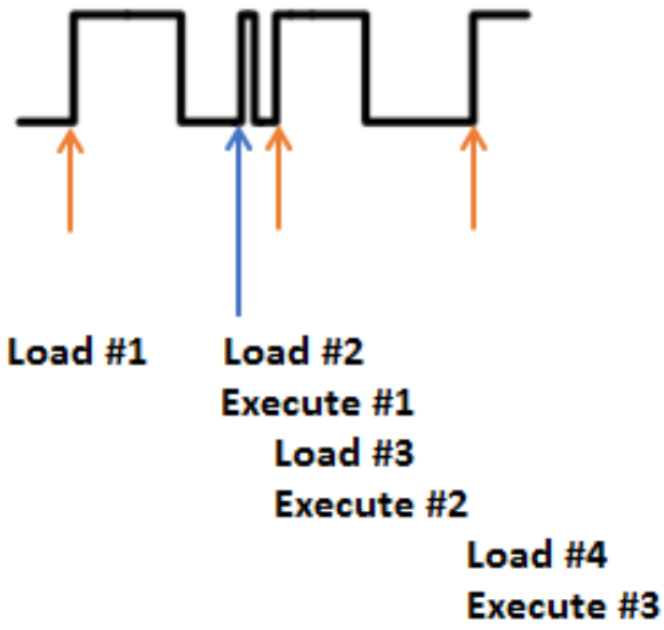


Figure 8.2: Device Clock With a Glitch

A drawback of clock glitching is that it will not work on devices that use internal oscillators or devices that have a phase lock loop (PLL) to derive a new clock from the external clock. High performance devices mostly fall into the latter category. [17]

8.2 Power Glitching

Power glitching targets the power supply of the device instead of the clock. Different methods of introducing glitches in this way have been proven to work, however the most temporally stable and therefore most precise one is through power spikes. Both positive and negative spikes result in similar waveforms internally in the target device. [17] Lowering the power supply also causes timing errors due to increased propagation delay, they are however difficult to predict, making it very hard to target specific instructions. Power glitching is generally more difficult to perform than clock glitching, requiring more precise glitch pulses. They can however target a wider range of devices, not requiring an external clock. Other glitching methods exist, e.g. electromagnetic (EM) glitching, also called EMFI, but they are not discussed here, since they are not supported by the ChipWhisperer tool.

8.3 Glitching the ZYNQ-7000 SoC

8.3.1 Test Server Setup

Before starting with the experiments, the Raspberry Pi test server had to be set up. For the operating system, an image of Raspbian Buster Lite was burned onto a microSD card and an empty file called “SSH” was added, to enable SSH by default. After realizing that the switching

regulators on the board may be interfering with the Pi's WiFi chip, an internet connection over Ethernet was established. After replacing the default password and updating the system, a VNC client was installed to provide both team members with access to the test server. Finally, the ChipWhisperer tool-chain was installed via pip.
To get a serial shell to the ZYNQ, install the "screen" package and run the "raspi-config" command as superuser. In the interfacing options, disable the "shell over serial" option and enable the serial port hardware. The serial terminal can now be started with the following command:

```
screen /dev/ttyS0 115200
```

8.3.2 ZYNQ Drivers and Firmware

An experiment requires two software parts, the Python script that controls the ChipWhisperer, and the firmware that controls the ZYNQ. This section describes the process of building these drivers and firmware in the Xilinx Vivado toolchain and the software development kit (SDK).
Start Vivado and create a new RTL project for the part number XC7Z007SCLG400-1. If we were using a well-known development board, we could choose the name of the board instead of the device, because a board definition file would already be available. The board definition file describes the available peripherals on the board and their connections. Luckily, we don't need to write our own, because we are not using the programmable logic, and the connections to the MIO pins are not really flexible, so we will be sticking to the datasheet.
Create a new block design and add a ZYNQ PS IP block. Select "customize IP" and load the custom preset "zbib_ps_setup.tcl" from our source folder. This is a preset script we exported from our settings and should work to configure the PS correctly. The differences to the default setup are:

- Bank1 set to 3.3V
- Ethernet0 disabled
- USB0 disabled
- MIO config: disable SDIO write protect (WP), remove USB and I2C reset
- Clock config: CPU 50MHz with ratio 4:2:1, DDR 200MHz, QSPI 40MHz
- DDR config: disable DDR

The PS configuration should now look like in figure 8.3.

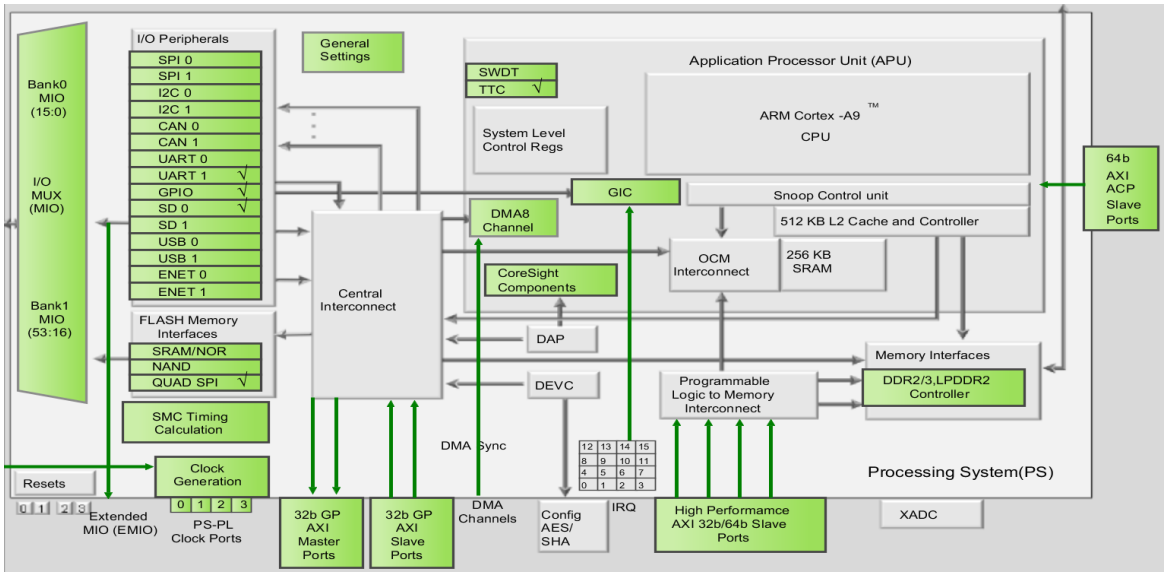


Figure 8.3: ZYNQ Processing System configuration

Next, run design automation for "FIXED_IO" and connect "M_AXI_GP0_ACLK" to "FCLK_CLK0" to connect the MIO subsystem and supply the AXI bus with a clock signal. The resulting block design should look like in figure 8.4.

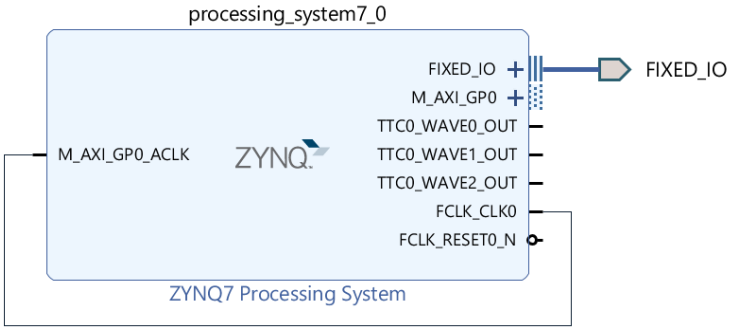


Figure 8.4: ZYNQ Block Design

Now select "create HDL wrapper" on "base.bd" and then generate the bitstream. This might take a while. The hardware platform can now be exported to the SDK with "export hardware", including the bitstream and the hardware definition file. It contains information about the available chip peripherals and their addresses, which are needed for the drivers.
Launch the SDK and create a new board support package and application with the hello world template. Build it, then select "program FPGA" and then "run on hardware". The JTAG debugger will connect to the ZYNQ, program it and execute the hello world program.
To load the program from a storage device like the SPI flash or the SD card, a first-stage bootloader (FSBL) is required. There is a standard implementation by Xilinx, but it needs to be slightly modified to work with our DDR-less system, as described in this article by Trenz Electronics. In main.c of the FSBL, the DDR RAM initialization and check needs to be disabled according to listing 8.1.

```
296 #define XPAR_PS7_DDR_0_S_AXI_BASEADDR 0
297 #ifdef XPAR_PS7_DDR_0_S_AXI_BASEADDR
```

```
298
299     /*
300      * DDR Read/write test
301      */
302     //Status = DDRInitCheck();
303     if (Status == XST_FAILURE) {
304         fsbl_printf(DEBUG_GENERAL,"DDR_INIT_FAIL \r\n");
305         /* Error Handling here */
306         OutputStatus(DDR_INIT_FAIL);
307         /*
308          * Calling FsblHookFallback instead of Fallback
309          * since, devcfg driver is not yet initialized
310          */
311         FsblHookFallback();
312     }
```

Source Code 8.1: FSBL: main.c (Rev. 16.00a)

To load both the FSBL and the application into the on-chip memory (OCM), a new linker script needs to be generated. Put the FSBL at the "AXI_RAM_0" address and the application at the "AXI_RAM_1" address. Now we need to also change the part in the FSBL code where the loading address of the application is checked. Loading from the OCM would trigger a fallback due to it being an illegal loading address, so it needs to be disabled, according to listing 8.2.

```
416 /*
417  * Load address check
418  * Loop will break when PS load address zero and partition is
419  * un-signed or un-encrypted
420  */
421 if ((PSPartitionFlag == 1) && (PartitionLoadAddr < DDR_START_ADDR)) {
422     if ((PartitionLoadAddr == 0) &&
423             (!((SignedPartitionFlag == 1) ||
424                     (EncryptedPartitionFlag == 1)))) {
425         break;
426     } else {
427         fsbl_printf(DEBUG_GENERAL,
428                 "INVALID_LOAD_ADDRESS_FAIL\r\n");
429         OutputStatus(INVALID_LOAD_ADDRESS_FAIL);
430         //FsblFallback();
431     }
432 }
433
434 if (PSPartitionFlag && (PartitionLoadAddr > DDR_END_ADDR)) {
435     fsbl_printf(DEBUG_GENERAL,
436             "INVALID_LOAD_ADDRESS_FAIL\r\n");
437     OutputStatus(INVALID_LOAD_ADDRESS_FAIL);
438     //FsblFallback();
439 }
```

Source Code 8.2: FSBL: image_mover.c (Rev. 11.00a)

After building everything, we are now ready to generate a boot image. Load a .bif (boot image format) file from our sources or create your own. Put the FSBL first and the application second in order. Depending on whether the boot image should be stored in the SPI flash or on the SD card, a .mcs or a .bin file need to be generated.

8.3.3 Voltage Glitching

ZYNQ Target Code

For the glitching experiments, the ZYNQ is executing the target code, which consists mainly of triggering the ChipWhisperer and then looping and waiting to get glitched. The exact program flow is as follows:

- Configure GPIO subsystem and the trigger pin
- Reset the trigger pin
- Trigger the ChipWhisperer
- Loop for 1000 rounds and count
- At the end of the loop, check the state of the counter: If it is 1000, the loop exited naturally. If the counter is not 1000, it is very likely that a glitch caused a malformed instruction, breaking the processor out of the loop
- Output the result on the trigger pin and over the serial interface

Then the target will be reset and the next run is started. The complete target C code can be found in listing 11.1 in the appendix.

ZYNQ ChipWhisperer Interface

The electrical signals between the ZYNQ and the ChipWhisperer are defined in table 8.1 and are helpful in writing the glitching scripts.

CW pin 4	FPGA-HS1	ZYNQ clock	Input
CW pin 5	PROG-RESET	ZYNQ POR	Output
CW pin 6	FPGA-HS2	ZYNQ Clock	Output
CW pin 10	FPGA-TARG1	ZYNQ TXD	Input
CW pin 12	FPGA-TARG2	ZYNQ RXD	Output
CW pin 13	PROG-PDIC	ZYNQ SRST	Output
CW pin 14	FPGA-TARG3	ZYNQ INIT	Input
CW pin 15	PROG-PDID	ZYNQ DONE	Input
CW pin 16	FPGA-TARG4	LED USER	Input

Table 8.1: Communication and Trigger Connections Between ChipWhisperer And ZYNQ

ChipWhisperer Script

The program flow in the glitching script is as follows:

- Connect to the ChipWhisperer via USB
- Configure the glitching module
- Configure the trigger module
- Set initial glitching parameters and enter a loop:
- Reset the target
- Arm the glitching module and wait for the trigger
- The CW will wait the configured number of clock cycles and then inject the glitch pulse
- Check for the result in the serial Output
- If successful: Stop
- If unsuccessful: Adjust parameters and go back to the start of the loop

The complete glitching script can be found in listing 11.2.
Setting the initial parameters to a reasonable value is difficult, because they are different for every target. At the beginning, the easiest option is to use the maximum parameter range of the ChipWhisperer and do a relatively coarse run over a wide range. This will take some time, but the parameters can then be narrowed down if any glitches are found.
We encountered an unexpected problem with the voltage glitching mode, where often the ChipWhisperer glitch output would remain in a high level during glitching runs. This is dangerous, as it continuously activates the crowbar circuit that short the target core power rails. We found this issue before it caused us any grief, because we measured the glitch output with the oscilloscope before connecting it to the ZBIB. But we could not conduct the experiments like this, and we were unable to find a solution to this problem, so we proceeded with clock glitching instead.

8.3.4 Clock Glitching

The target code and ChipWhisperer scripts are essentially the same for clock glitching, except for a different glitching module and clock configuration in the latter. The updated script can be found in listing 11.3.
Because the ZYNQ is now receiving its system clock (and occasional glitches) from the ChipWhisperer instead of the on-board oscillator, the CW needs to be configured to output this signal. That's why the clock generator (CLKGEN) is set up in the new script.
The ZYNQ-7000 processor does not receive the clock signal applied to the system clock input directly, it is fed through a phase-locked loop (PLL). This PLL employs its own internal oscillator that is synchronized with the external clock. The downside in this case is that clock glitches applied to the system clock input do not propagate through to the processor. To fix this, the PLL needs to be bypassed. The switch on the ZBIB is set to the bypass position and a small adaptation to the code must be made again. By default, the FSBL waits for the PLL to lock, which will not happen if it is disabled and bypassed. Therefore, the loop that waits for the lock

needs to be removed in “ps7_init.c”.
Now we can finally run our glitching scripts. First we did a coarse run again with a single glitch pulse, the oscilloscope capture of the event can be seen in figure **??**. The upper trace is the system clock and the lower trace is the power consumption. As we can see, the processor does not seem to be affected, which was confirmed by the output of the target, which showed no malfunctions.

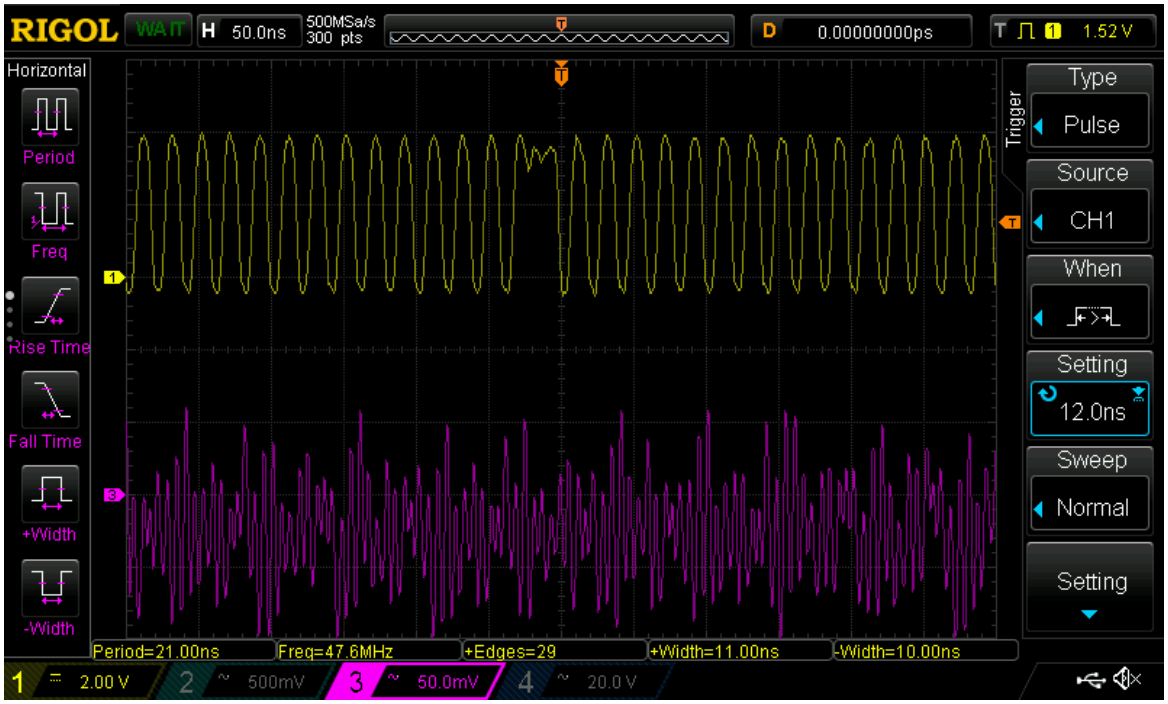


Figure 8.5: Single Clock Glitch Showing No Effect on the Power Consumption
Source: NewAE Technology, newae.com

Then we tried so inject multiple glitch pulses in a row to try and disturb the processor. The result can be seen in figure 8.6, where the power consumption clearly shows a difference from the normal operation. This was confirmed by the target serial output, which showed a few successful glitches over the run. The logs are available in the source folder.

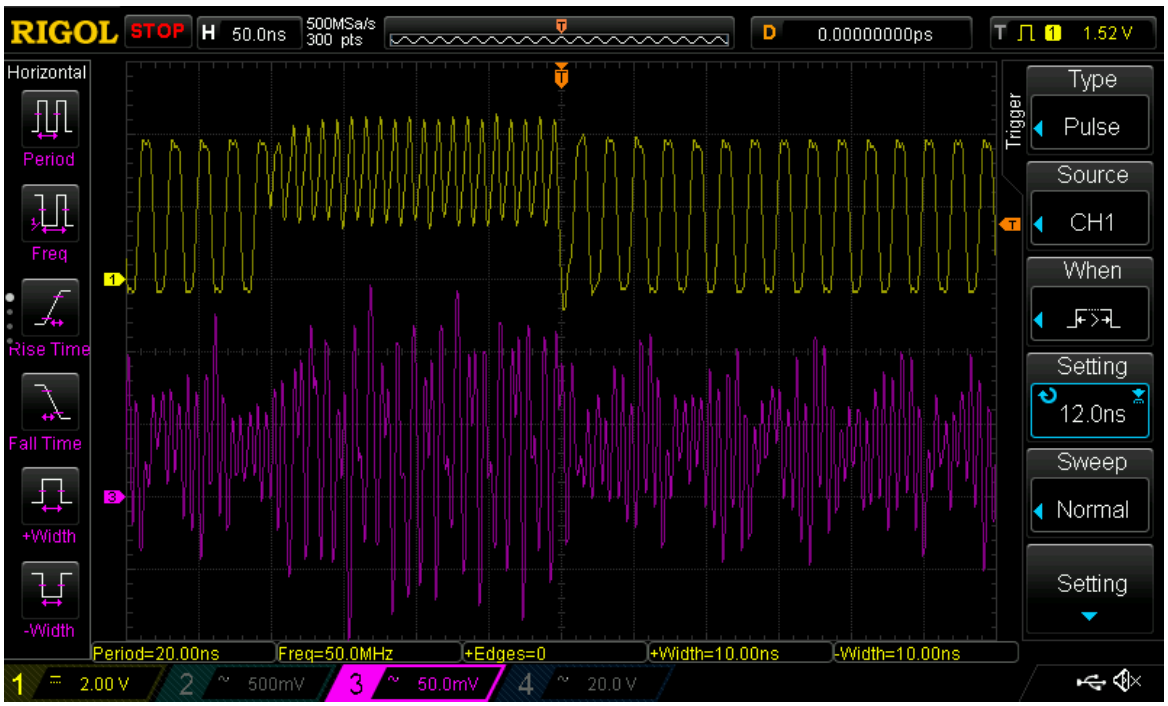


Figure 8.6: Multiple Clock Glitches Showing a Change in the Power Consumption
Source: NewAE Technology, newae.com

It was however difficult to narrow down the parameters because the glitches seemed to often occur with different parameters and not totally reliable. At this point we run out of time assigned to the glitching experiments and moved on the fuzzing techniques. For proceeding, the top priority would be to try again to narrow down the parameter range for more reliable glitches. After that, one could try to apply the findings to glitching the FSBL and eventually the boot ROM.

9 Fuzzing

Fuzzing is software testing method to find errors in a program by providing random, invalid or unexpected data as input during runtime in an automated fashion. Fuzzers work best to detect vulnerabilities that cause the system to crash. They are less useful in detecting other system threats because an error that does not cause the system to malfunction is difficult to detect. Fuzzers are useful in testing irregular inputs that might have been forgotten in regular testing (e.g. unit tests). [11] It can test the software from a black-box or grey-box environment and is therefore important from an attackers point of view. When testing secure boot on an embedded system an attacker would likely not have access to the source code of the BootROM.

Fuzzing can be separated into three steps:

- Step 1: Generate data
- Step 2: Provide data to software
- Step 3: Monitor the software to detect anomalies

9.1 Data Generation

There are two possibilities for data generation. One can either generate new data that is shaped to fit the input or modify existing data with random changes. Modifications on existing data are called mutations. In a black-box setting, meaning there is no knowledge about system internals, it is easiest to generate random data. Complete randomness in data generation has the broadest spectrum of faults it can detect, however it also is the most inefficient. By focusing the generated data on edge cases it is possible to significantly reduce the number of inputs that do not cause any unexpected behavior.

9.2 What To Attack?

In SD boot mode the boot image file is read from the SD card and copied to internal memory. This provides surface for attacks by fuzzing. One approach would be to try and fuzz the boot image itself by misrepresenting partition sizes or construction of malicious partitions.

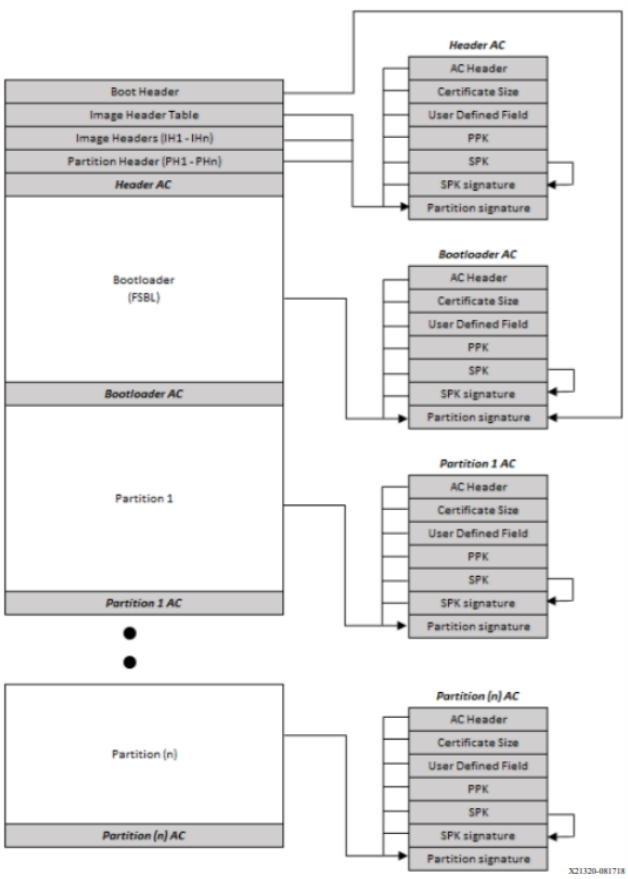


Figure 9.1: ZYNQ-7000 SoC Device Boot Image [3, p.20]

As 9.1 shows, when the basic secure boot guidelines are followed, each part of the boot image is signed and could prove quite difficult to manipulate. For this project this would have been outside the scope.

9.2.1 FAT

A different approach is to format the boot SD card to FAT and then fuzz the raw data on the boot volume. FAT is a small and simple file system, originally developed by Microsoft, that is understood by nearly all operating systems and therefore a common choice for firmware-based projects. [21]

The figure below shows the basic structure of a FAT volume.

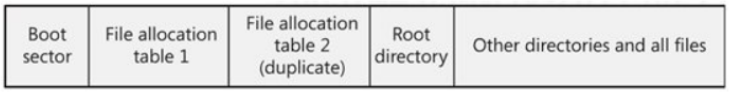


Figure 9.2: FAT Volume Organization [10]

The first section of a FAT drive is called the boot sector. It contains general information about the file system itself (e.g. number of bytes per sector and number of sectors per cluster). The

boot sector is followed by the File Allocation Tables (FATs). The drive is divided into clusters, with the number of sectors per cluster listed in the boot sector byte 13. The file allocation table contains one entry per cluster, which uses 28 bits for FAT32. [5]

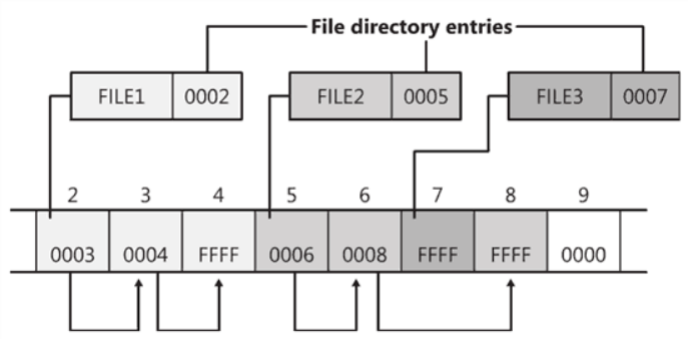


Figure 9.3: FAT Allocation Chain Example [10]

Each file directory entry contains a number that corresponds to an entry in the FAT. The entry linked by the directory entry represents the starting cluster of the file. The FAT entry at the corresponding location contains either the number of the next cluster or the reserved value of 0xFFFFFFFF for FAT32, meaning the end of the file is this cluster. A value of zero in the FAT table indicates an unused cluster.

Following the FATs comes the root directory. The only difference between a regular directory and the root directory is that it has a fixed size and fixed location on the drive. After that are the clusters with all other directories and files. Each directory entry has the structure shown in figure 9.4.

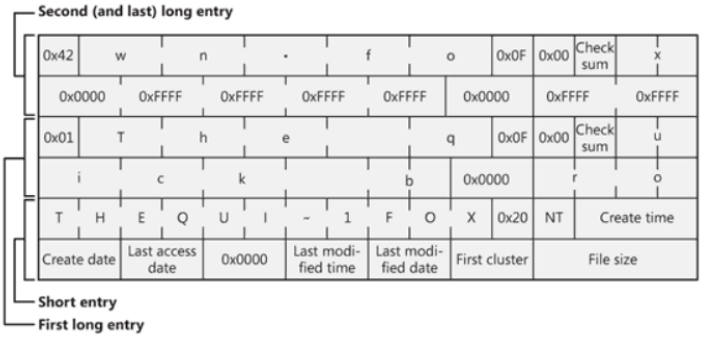


Figure 9.4: FAT Directory Entry Structure [10]

9.2.2 Fuzzing Attack

In case the FAT implementation used by Xilinx contains weaknesses it is plausible one could discover them by fuzzing the boot drive. Our approach was therefore to manipulate certain fields in the raw data of the SD card. By manipulating file size and the FAT itself it may be possible to inject code or extract information from the system. Due to time constraints only a basic testing setup was prepared for the ZYNQ-7000 SoC without any physical tests being run. The fuzzer implementation is a simple black box fuzzer, since we have no information about the boot ROM internals. It is a randomizer that changes the size value of the directory entry for the

boot binary file as a first step. For connecting to the drive with intent to manipulate the raw data, the PySerial library could be used. [URL: https://pyserial.readthedocs.io/en/latest/pyserial_api.html (visited on 06/11/2020)]

```
import serial
with serial.Serial('/dev/ttyS1', 19200, timeout=1000) as ser:
        s = ser.read(10000)
```

The offset of the target field was determined manually by finding the directory entry and calculating the offset from that point. The file size value is then changed to a random value and written back to the boot drive.

```
ser.write(s)    # write the manipulated string back to the drive.
```

Should parts of the boot ROM code become known, that should allow for more precise targeting in the fuzzing logic making the fuzzer more efficient.

9.2.3 Fuzzing Automation

The Raspberry Pi is used to control not only the fuzzing process itself, but also automates the deployment to the ZYNQ via the SD card and multiplexer. The process for a fuzzing run is as follows:

- Set the ZYNQ reset pin (POR)
- Switch the SD card via the multiplexer to the Raspberry Pi
- Wait for the SD card to be mounted
- Transfer the new fuzzing image to the SD card and unmount
- Disable power to the SD card and wait a short amount of time
- Switch the SD card via the multiplexer to the ZYNQ
- Enable power to the SD card
- Release the ZYNQ reset pin
- Read the run result from the GPIO interface to the ZYNQ

This process is then repeated as many times as necessary. A bash script using the “gpio” utility can be used to control the I/O pins. The SD card was not recognized in the beginning, because it is only connected to the secondary SDIO interface on the Raspberry Pi, but this was fixed by adding an SDIO overlay to the device tree.

9.3 Error Detection

For our setup, detection of the fuzzing results proved a big challenge. With a standard secure boot setup the JTAG interface is completely disabled, preventing us from having any meaningful output to interpret. The system itself represent a black-box with no obvious way to access any

internal information. The most basic and only obvious way of detecting system state after a fuzzing attempt is the status LED.

10 Conclusion and Outlook

10.1 Commentary

«««< HEAD This study was conducted to synthesize a test suite for embedded systems. It was a follow up to our semester project "Secure Boot on Embedded Systems". It is a compilation of tests to evaluate the robustness of an embedded system against the most important hacking methods used against such systems. ======= This study was conducted to elaborate a test suite for embedded systems and was a follow up to our semester project "Secure Boot on Embedded Systems". It is a compilation of tests to evaluate the robustness of an embedded system against the most important hacking methods targeting such systems. »»»> 34cbd7b1f34f37387b8ad5e1631c698dc863d9e8

10.2 Conclusion

The custom evaluation board developed for this project creates an ideal testing environment for physical hacking methods performed with the ChipWhisperer tool. It is kept as simple as possible, while providing all necessary functionality, to make its design adaptable to as many different microprocessors as possible. The compiled test suite covers the most important hacking methods in today's embedded industry. It includes a guide for code analysis and highlights some of the most common logical vulnerabilities in embedded system code.
A successful power analysis attack on AES-128 was performed on an XMEGA target and the test should be usable on a wide variety of systems with minimal adaptation work. A successful clock glitching attack was performed on the ZYNQ-7000 SoC, showing that it is in principle possible to manipulate code flow of the CPU leading to vulnerabilities against the mentioned attacks. The tests can be adapted for different glitching methods, e.g. voltage glitching. Fuzzing experiments were designed to illustrate the process of designing a fuzzer for a black-box embedded system.

10.3 Restrictions

The lockdown due to the COVID-19 pandemic necessitated the construction of a lab setup that allowed for remote access to all relevant parts of the testing environment. Additional work caused by this situation forced us to reduce the scope of parts of the fuzzing experiments as well as the power analysis tests. The power analysis attack on AES was performed on the target board included in the ChipWhisperer-Lite purchase instead of our custom evaluation board and therefore on the XMEGA micro controller instead of the ZYNQ-7000 SoC. Due to time constraints the fuzzing setup was not completed entirely. The fuzzing logic to manipulate FAT formatted drive was tested on a USB drive instead of the target SD card and requires more work to be used on the target setup.

10.4 Outlook

«««< HEAD This test suite could be expanded on indefinitely and an embedded device manufacturer would need to identify the biggest threats to the system they want to design. Most of these tests could be integrated into a development pipeline without excessive amounts of work.
Code Analysis is already a standard practice for any software development and most manufacturers already have clearly defined coding guidelines.
The power analysis experiments should be performed on our target system in the context of the evaluation board to verify the portability of the attack. Performing more sophisticated attacks, e.g. correlation power analysis, would only be necessary if differential power analysis does not show the expected results.
Since the target was successfully glitched, a logical next step would be to try and exploit the system using the glitch, e.g. by bypassing authentication or manipulating specific machine code instructions to exploit the system in another way. It would also be desirable to perform a power glitch on the system allowing to circumvent the PLL present on the board
The fuzzing software needs to be tested on the target system and the fuzzer itself could be used to attack different parts of the system e.g. the boot image file. Additional knowledge about system internals could enable more sophisticated fuzzing strategies and could make it more likely to find exploitable vulnerabilities. ======= This test suite could be expanded almost indefinitely and an embedded device manufacturer would need to identify the biggest threats to the system they want to design. Most of these tests could be integrated into a development pipeline without an excessive amount of work.
Code Analysis is already a standard practice for any software development and most manufacturers already have clearly defined coding guidelines.
The power analysis experiments should be performed on our target system in the context of the evaluation board to verify the portability of the attack. Performing more sophisticated attacks, e.g. correlation power analysis, would only be necessary if differential power analysis does not show the expected results.
Since we successfully glitched the target, a logical next step would be to try and exploit the system using the glitch, e.g. by bypassing authentication or manipulating specific machine code instructions to exploit the system in another way. It would also be desirable to perform a power glitch on the system allowing to circumvent the PLL present on the board that poses an obstacle to clock glitching attacks.
The fuzzing software needs to be tested on the target system and the fuzzer itself could be used to attack different parts of the system, e.g. the boot image file. Additional knowledge about system internals could enable more sophisticated fuzzing strategies and could make it more likely to find exploitable vulnerabilities. »»»> 34cbd7b1f34f37387b8ad5e1631c698dc863d9e8

10.5 Acknowledgments

We would like to thank Prof. Stefan Richter for his guidance and for enabling us to work on this fascinating project and for encouraging and supporting us during its progression. We also want to thank Florian Bruhin for providing us with the LaTeX template used to write this thesis, as well as all other related documents.

11 List of Abbreviations

ADC Analog-to-Digital Converter

AES Advanced Encryption Standard

ARM Advanced RISC Machine

AXI Advanced eXtensible Interface

BBRAM Battery-Backed Random Access Memory

BGA Ball Grid Array

BIF Boot Image Format

BSP Board Support Package

CAD Computer-Aided Design

CBC Cipher Block Chaining

CPA Correlation Power Analysis

CPU Central Processing Unit

CRC Cyclic Redundancy Check

DC/DC Direct Current to Direct Current

DDR Double Data Rate

DPA Differential Power Analysis

DRC Design Rule Check

DUT Device Under Test

ECU Electronic Control Unit

EMFI Electromagnetic Fault Injection

EMIO Extended Multiplexed Input/Output

FAT File Allocation Table

FI Fault Injection

FPGA Field-Programmable Gate Array

FSBL First Stage Boot Loader

FW Firmware

GPIO General Purpose Input Output

HMAC Hash-based Message Authentication Code

HSM Hardware Security Module

I2C Inter-Integrated Circuit

IC Integrated Circuit

I/O Input/Output

IoT Internet of Things

IP Intellectual Property

JTAG Joint Test Action Group

LAN Local Area Network

LDO Low-Dropout Regulator

LED Light-Emitting Diode

LNA Low-Noise Amplifier

MIO Multiplexed Input/Output

NVM Non Volatile Memory

OCM On-Chip Memory

OTP One Time Programmable

PCAP Processor Configuration Access Port

PCB Printed Circuit Board

PG Power Good

PKI Public Key Infrastructure

PL Programmable Logic

PLL Phase-Locked Loop

Pmod Peripheral Module

PoC Proof of Concept

POR Power-On Reset

PPK Primary Public Key

PS Processing System

PSK Primary Secret Key

PSRR Power Supply Rejection Ratio

RAM Random-Access Memory

RISC Reduced Instruction Set Computer

ROM Read-Only Memory

ROT Root of Trust

RSA Rivest-Shamir-Adleman

SD Secure Digital

SDIO Secure Digital Input Output

SDK Software Development Kit

SHA Secure Hashing Algorithm

SMD Surface-Mounted Device

SNR Signal-to-Noise Ratio

SoC System on Chip

SPI Serial Peripheral Interface

SPK Secondary Public Key

SSBL Second Stage Boot Loader

SSK Secondary Secret Key

TP Test Point

UART Universal Asynchronous Receiver-Transmitter

USB Universal Serial Bus

WP Write Protect

ZBIB Zynq Break-In Board

Bibliography

[1] et al. A. Barenghi. *Low Voltage Fault Attacks on the RSA Cryptosystem*. 2009.

[2] et al. Alessandro Barenghi. *Fault Injection Attacks on Cryptographic Devices: Theory, Practice and Countermeasures*. Nov. 1, 2012.

[3] *Bootgen User Guide*. User Guide 1283. Version v2018.2. Xilinx. Sept. 28, 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1283-bootgen-user-guide.pdf.

[4] Olivier F. Brier E. Clavier C. *Correlation Power Analysis with a Leakage Model*. 2006.

[5] Andries Brouwer. *FAT*. Sept. 20, 2002. URL: https://www.win.tue.nl/~aeb/linux/fs/fat/fat-1.html (visited on 06/04/2020).

[6] et al. Bruce McCorkendale. *Systems And Methods For Combining Static And Dynamic Code Analysis*. May 13, 2014, p. 6.

[7] Drew Buttner. *The importance of manual secure code review*. Mitre. Jan. 16, 2014. URL: https://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/the-importance-of-manual-secure-code-review (visited on 03/17/2020).

[8] Francesco Palmarini Claudio Bozzato Riccardo Focardi. *Shaping the Glitch: Optimizing Voltage Fault Injection Attacks*. 2019.

[9] A. Shamir E. Biham. *Differential fault analysis of secret key cryptosystems*. 1997.

[10] Network Encylopedia. *File Allocation Table (FAT)*. URL: https://networkencyclopedia.com/file-allocation-table-fat/ (visited on 05/14/2020).

[11] kingthorin. "Fuzzing". In: *OWASP* (Apr. 24, 2020). URL: https://owasp.org/www-community/Fuzzing (visited on 06/04/2020).

[12] Jun B. Kocher P. Jaffe J. "Differential Power Analysis". In: *CRYPTO 1999: Advances in Cryptology* (1999), pp. 388–397.

[13] Gary Robinson Larry Conklin. *OWASP Code Review Guide 2.0*. Version RELEASE. OWASP. URL: https://owasp.org/www-pdf-archive/OWASP_Code_Review_Guide_v2.pdf (visited on 04/20/2020).

[14] et al. Le TH. *A Proposition for Correlation Power Analysis Enhancement*. 2006.

[15] P. Louridas. "Static code analysis". In: *IEEE Software* 23 (2006), pp. 58–61.

[16] Stefan Mangard. “Differential Power Analysis”. In: *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Boston, MA: Springer US, 2007, pp. 119–165. ISBN: 978-0-387-38162-6.

[17] Colin O’Flynn. *Fault Injection using Crowbars on Embedded Systems*. 2016.

[18] G. Letourneux P. Dusart and O. Vivolo. *Differential fault analysis on A.E.S.* 2003.

[19] Andreas Reiter. *In-Memory Fuzzing on Embedded Systems*. Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, 2012.

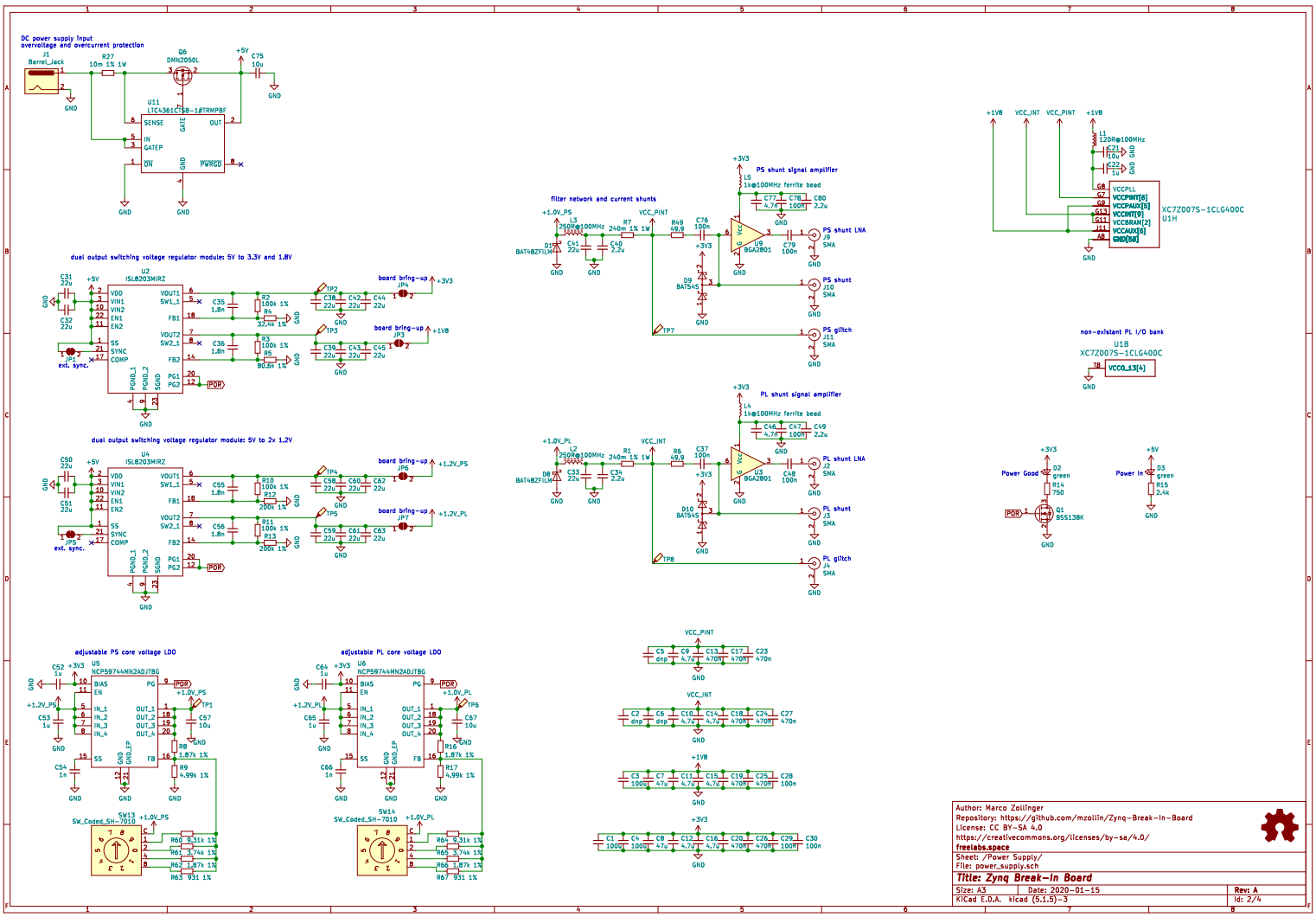
[20] P. Socha, J. Brejník, and M. Bartik. “Attacking AES implementations using correlation power analysis on ZYBO Zynq-7000 SoC board”. In: *2018 7th Mediterranean Conference on Embedded Computing (MECO)*. 2018, pp. 1–4.

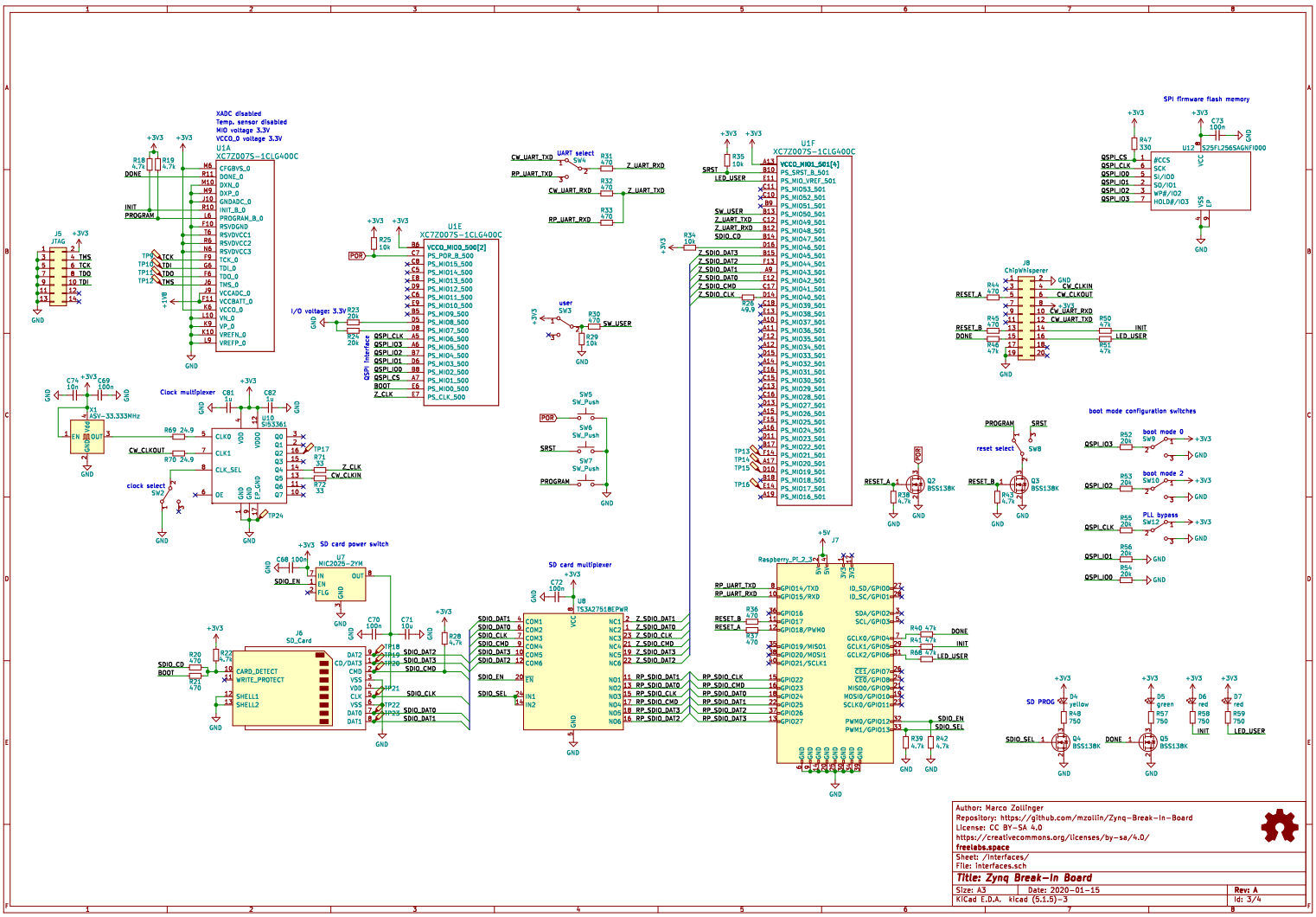
[21] Paul Stoffregen. *Understanding FAT32 Filesystems*. Feb. 24, 2005. URL: `http://www.pjrc.com/tech/8051/ide/fat32.html` (visited on 05/28/2020).

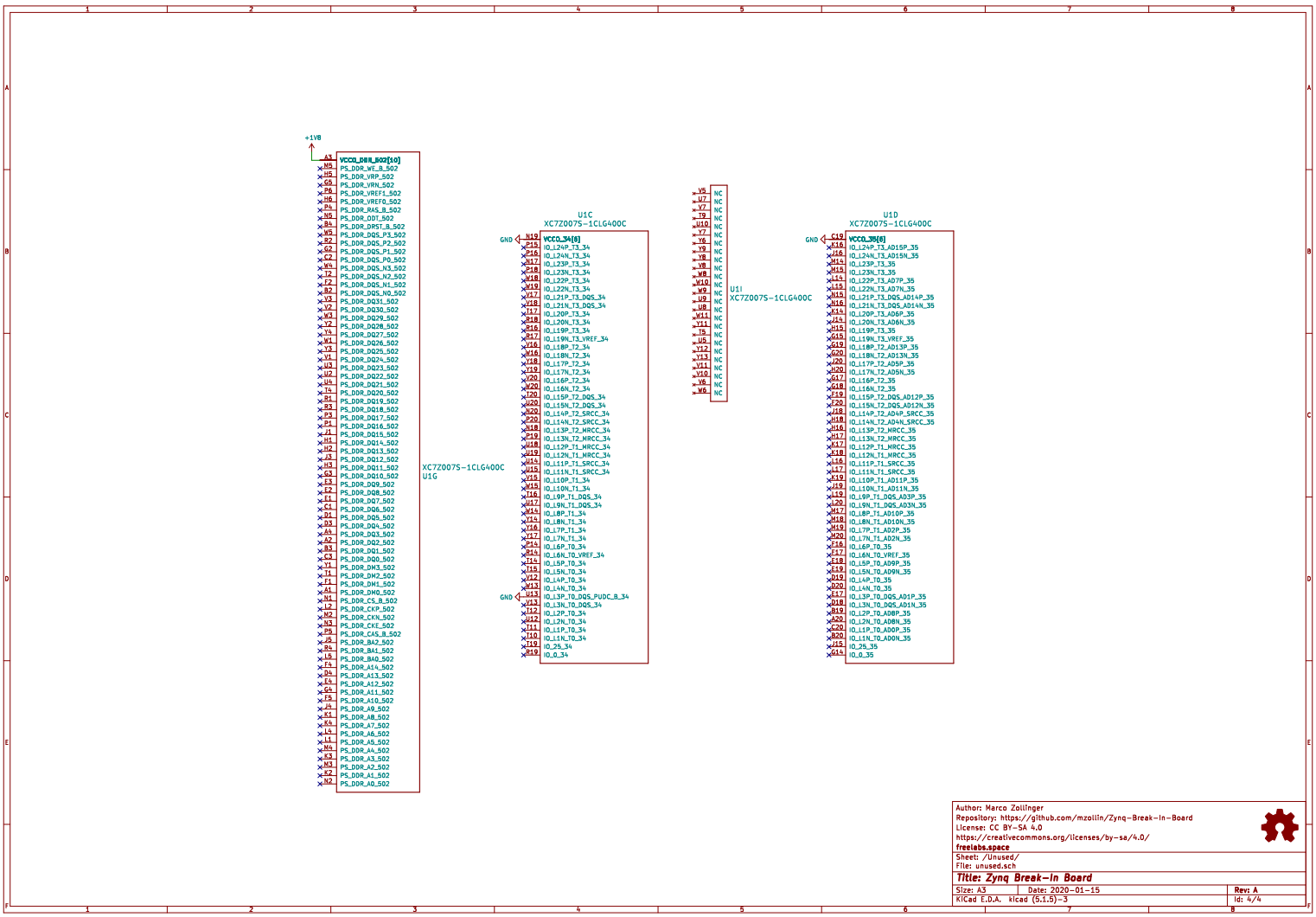
[22] CERN Computer Security Team. *Common vulnerabilities guide for C programmers*. URL: `https://security.web.cern.ch/recommendations/en/codetools/c.shtml` (visited on 05/30/2020).

Appendices

Zynq Break-In Board Schematics


DC power supply input
overvoltage and overcurrent protection
dual output switching voltage regulator module: 5V to 3.3V and 1.8V
dual output switching voltage regulator module: 5V to 2x 1.2V
filter network and current shunts
PS shunt signal amplifier
PL shunt signal amplifier
adjustable PS core voltage LDO
adjustable PL core voltage LDO
non-existant PL I/O bank
XC7Z007S-1CLG400C
Author: Marco Zollinger
Repository: https://github.com/mzollin/Zynq-Break-In-Board
License: CC BY-SA 4.0
https://creativecommons.org/licenses/by-sa/4.0/
freelabs.space
Sheet: /Power Supply/
File: power_supply.sch
Title: Zynq Break-In Board
Size: A3
Date: 2020-01-15
Rev: A
KiCad E.D.A. kicad (5.1.5)-3
Id: 2/4


XADC disabled
Temp. sensor disabled
MIO voltage 3.3V
VCCO_0 voltage 3.3V
U1A
XC7Z007S-1CLG400C
U1E
XC7Z007S-1CLG400C
U1F
XC7Z007S-1CLG400C
SPI firmware flash memory
U12
S25FL256SAGNFI000
J5
JTAG
J8
ChipWhisperer
Clock multiplexer
U10
5I53361
SD card power switch
U7
MIC2025-2YM
J6
SD_Card
SD card multiplexer
U8
TS3A27518EPWR
J7
Raspberry_Pi_2_3
boot mode configuration switches
boot mode 0
boot mode 2
PLL bypass
UART select
user
I/O voltage 3.3V
QSPI Interface
reset select
SD PROG
Author: Marco Zalinger
Repository: https://github.com/mzalin/Zynq-Break-In-Board
License: CC BY-SA 4.0
https://creativecommons.org/licenses/by-sa/4.0/
freelabs.space
Sheet: /Interfaces/
File: Interfaces.sch
Title: Zynq Break-In Board
Size: A3
Date: 2020-01-15
Rev: A
KiCad E.D.A. kicad (5.1.5)-3
Id: 3/4


+1V8
XC7Z007S-1CLG400C
U1G
VCCO_DDR_502[10]
PS_DDR_WE_B_502
PS_DDR_VRP_502
PS_DDR_VRN_502
PS_DDR_VREF1_502
PS_DDR_VREF0_502
PS_DDR_RAS_B_502
PS_DDR_ODT_502
PS_DDR_DRST_B_502
PS_DDR_DQS_P3_502
PS_DDR_DQS_P2_502
PS_DDR_DQS_P1_502
PS_DDR_DQS_P0_502
PS_DDR_DQS_N3_502
PS_DDR_DQS_N2_502
PS_DDR_DQS_N1_502
PS_DDR_DQS_N0_502
PS_DDR_DQ31_502
PS_DDR_DQ30_502
PS_DDR_DQ29_502
PS_DDR_DQ28_502
PS_DDR_DQ27_502
PS_DDR_DQ26_502
PS_DDR_DQ25_502
PS_DDR_DQ24_502
PS_DDR_DQ23_502
PS_DDR_DQ22_502
PS_DDR_DQ21_502
PS_DDR_DQ20_502
PS_DDR_DQ19_502
PS_DDR_DQ18_502
PS_DDR_DQ17_502
PS_DDR_DQ16_502
PS_DDR_DQ15_502
PS_DDR_DQ14_502
PS_DDR_DQ13_502
PS_DDR_DQ12_502
PS_DDR_DQ11_502
PS_DDR_DQ10_502
PS_DDR_DQ9_502
PS_DDR_DQ8_502
PS_DDR_DQ7_502
PS_DDR_DQ6_502
PS_DDR_DQ5_502
PS_DDR_DQ4_502
PS_DDR_DQ3_502
PS_DDR_DQ2_502
PS_DDR_DQ1_502
PS_DDR_DQ0_502
PS_DDR_DM3_502
PS_DDR_DM2_502
PS_DDR_DM1_502
PS_DDR_DM0_502
PS_DDR_CS_B_502
PS_DDR_CKP_502
PS_DDR_CKN_502
PS_DDR_CKE_502
PS_DDR_CAS_B_502
PS_DDR_BA2_502
PS_DDR_BA1_502
PS_DDR_BA0_502
PS_DDR_A14_502
PS_DDR_A13_502
PS_DDR_A12_502
PS_DDR_A11_502
PS_DDR_A10_502
PS_DDR_A9_502
PS_DDR_A8_502
PS_DDR_A7_502
PS_DDR_A6_502
PS_DDR_A5_502
PS_DDR_A4_502
PS_DDR_A3_502
PS_DDR_A2_502
PS_DDR_A1_502
PS_DDR_A0_502
U1C
XC7Z007S-1CLG400C
GND
VCCO_34[6]
IO_L24P_T3_34
IO_L24N_T3_34
IO_L23P_T3_34
IO_L23N_T3_34
IO_L22P_T3_34
IO_L22N_T3_34
IO_L21P_T3_DQS_34
IO_L21N_T3_DQS_34
IO_L20P_T3_34
IO_L20N_T3_34
IO_L19P_T3_34
IO_L19N_T3_VREF_34
IO_L18P_T2_34
IO_L18N_T2_34
IO_L17P_T2_34
IO_L17N_T2_34
IO_L16P_T2_34
IO_L16N_T2_34
IO_L15P_T2_DQS_34
IO_L15N_T2_DQS_34
IO_L14P_T2_SRCC_34
IO_L14N_T2_SRCC_34
IO_L13P_T2_MRCC_34
IO_L13N_T2_MRCC_34
IO_L12P_T1_MRCC_34
IO_L12N_T1_MRCC_34
IO_L11P_T1_SRCC_34
IO_L11N_T1_SRCC_34
IO_L10P_T1_34
IO_L10N_T1_34
IO_L9P_T1_DQS_34
IO_L9N_T1_DQS_34
IO_L8P_T1_34
IO_L8N_T1_34
IO_L7P_T1_34
IO_L7N_T1_34
IO_L6P_T0_34
IO_L6N_T0_VREF_34
IO_L5P_T0_34
IO_L5N_T0_34
IO_L4P_T0_34
IO_L4N_T0_34
IO_L3P_T0_DQS_PUDC_B_34
IO_L3N_T0_DQS_34
IO_L2P_T0_34
IO_L2N_T0_34
IO_L1P_T0_34
IO_L1N_T0_34
IO_25_34
IO_0_34
U1
XC7Z007S-1CLG400C
NC
U1D
XC7Z007S-1CLG400C
GND
VCCO_35[6]
IO_L24P_T3_AD15P_35
IO_L24N_T3_AD15N_35
IO_L23P_T3_35
IO_L23N_T3_35
IO_L22P_T3_AD7P_35
IO_L22N_T3_AD7N_35
IO_L21P_T3_DQS_AD14P_35
IO_L21N_T3_DQS_AD14N_35
IO_L20P_T3_AD6P_35
IO_L20N_T3_AD6N_35
IO_L19P_T3_35
IO_L19N_T3_VREF_35
IO_L18P_T2_AD13P_35
IO_L18N_T2_AD13N_35
IO_L17P_T2_AD5P_35
IO_L17N_T2_AD5N_35
IO_L16P_T2_35
IO_L16N_T2_35
IO_L15P_T2_DQS_AD12P_35
IO_L15N_T2_DQS_AD12N_35
IO_L14P_T2_AD4P_SRCC_35
IO_L14N_T2_AD4N_SRCC_35
IO_L13P_T2_MRCC_35
IO_L13N_T2_MRCC_35
IO_L12P_T1_MRCC_35
IO_L12N_T1_MRCC_35
IO_L11P_T1_SRCC_35
IO_L11N_T1_SRCC_35
IO_L10P_T1_AD11P_35
IO_L10N_T1_AD11N_35
IO_L9P_T1_DQS_AD3P_35
IO_L9N_T1_DQS_AD3N_35
IO_L8P_T1_AD10P_35
IO_L8N_T1_AD10N_35
IO_L7P_T1_AD2P_35
IO_L7N_T1_AD2N_35
IO_L6P_T0_35
IO_L6N_T0_VREF_35
IO_L5P_T0_AD9P_35
IO_L5N_T0_AD9N_35
IO_L4P_T0_35
IO_L4N_T0_35
IO_L3P_T0_DQS_AD1P_35
IO_L3N_T0_DQS_AD1N_35
IO_L2P_T0_AD8P_35
IO_L2N_T0_AD8N_35
IO_L1P_T0_AD0P_35
IO_L1N_T0_AD0N_35
IO_25_35
IO_0_35
Author: Marco Zallinger
Repository: https://github.com/mzallin/Zynq-Break-In-Board
License: CC BY-SA 4.0
https://creativecommons.org/licenses/by-sa/4.0/
freelabs.space
Sheet: /Unused/
File: unused.sch
Title: Zynq Break-In Board
Size: A3
Date: 2020-01-15
Rev: A
KiCad E.D.A. kicad (5.1.5)-3
Id: 4/4

Zynq Break-In Board Layout: Layer by Layer

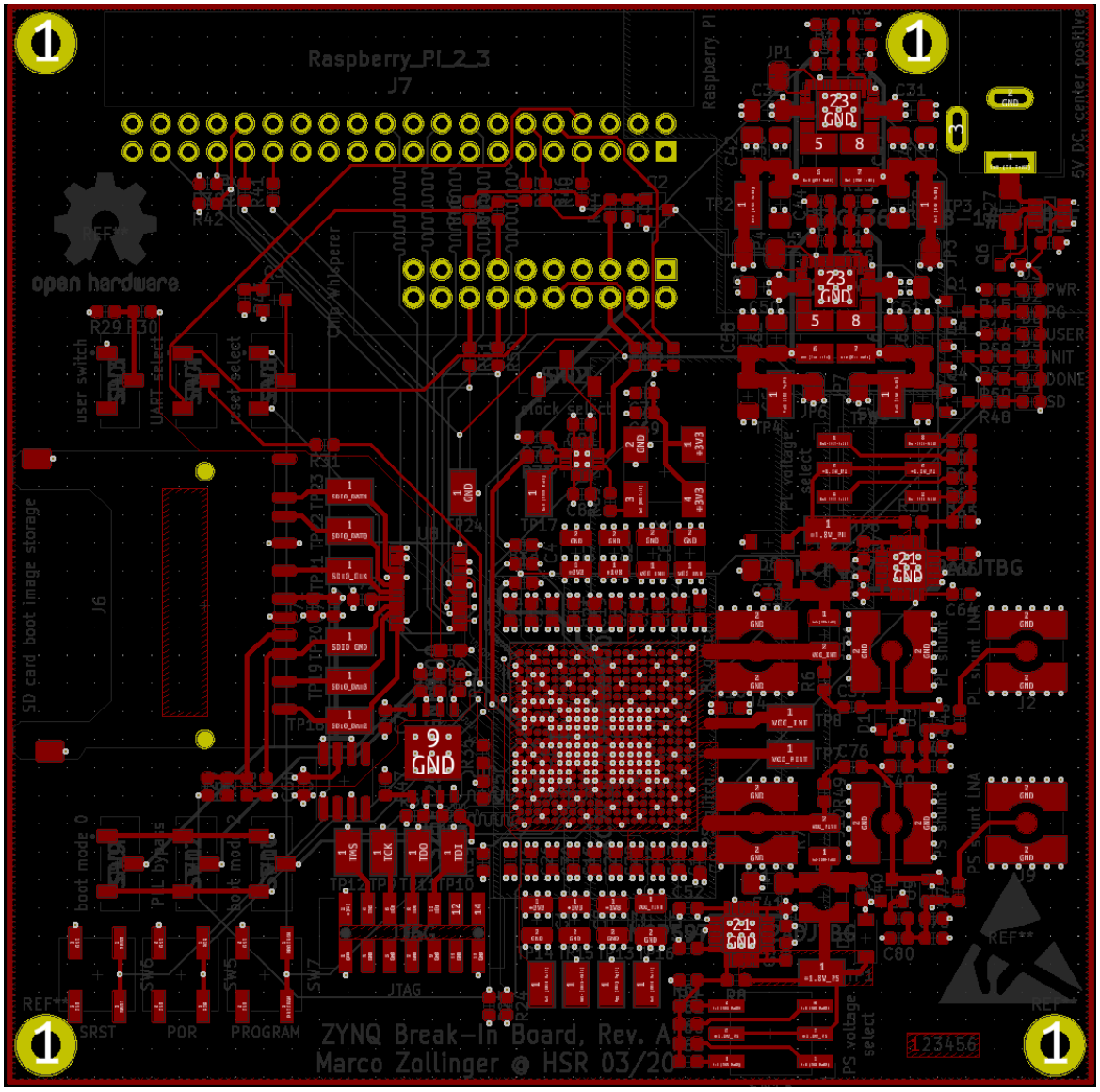


Figure 11.1: PCB Layer 1 (Top)

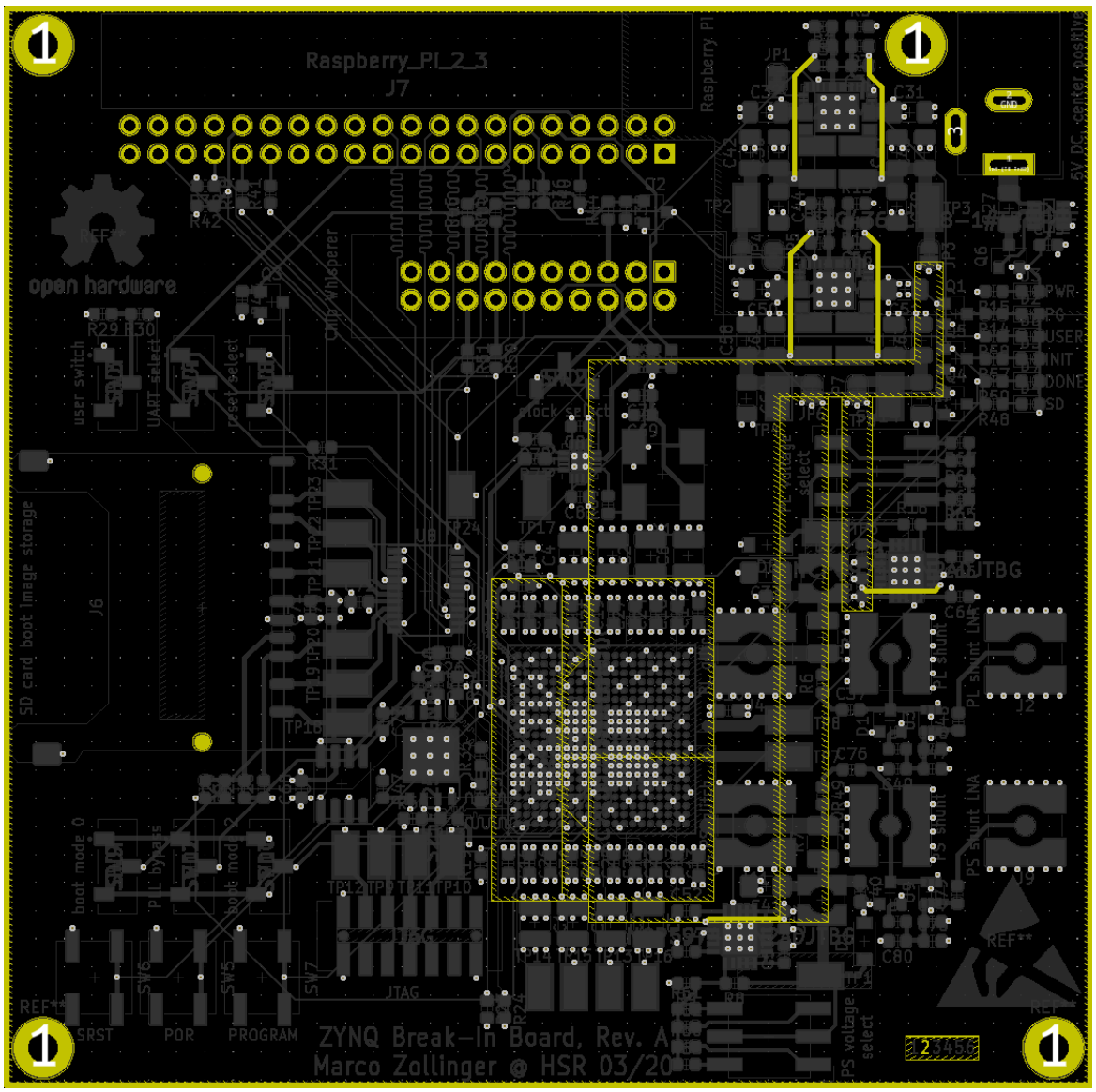


Figure 11.2: PCB Layer 2 (Inner)

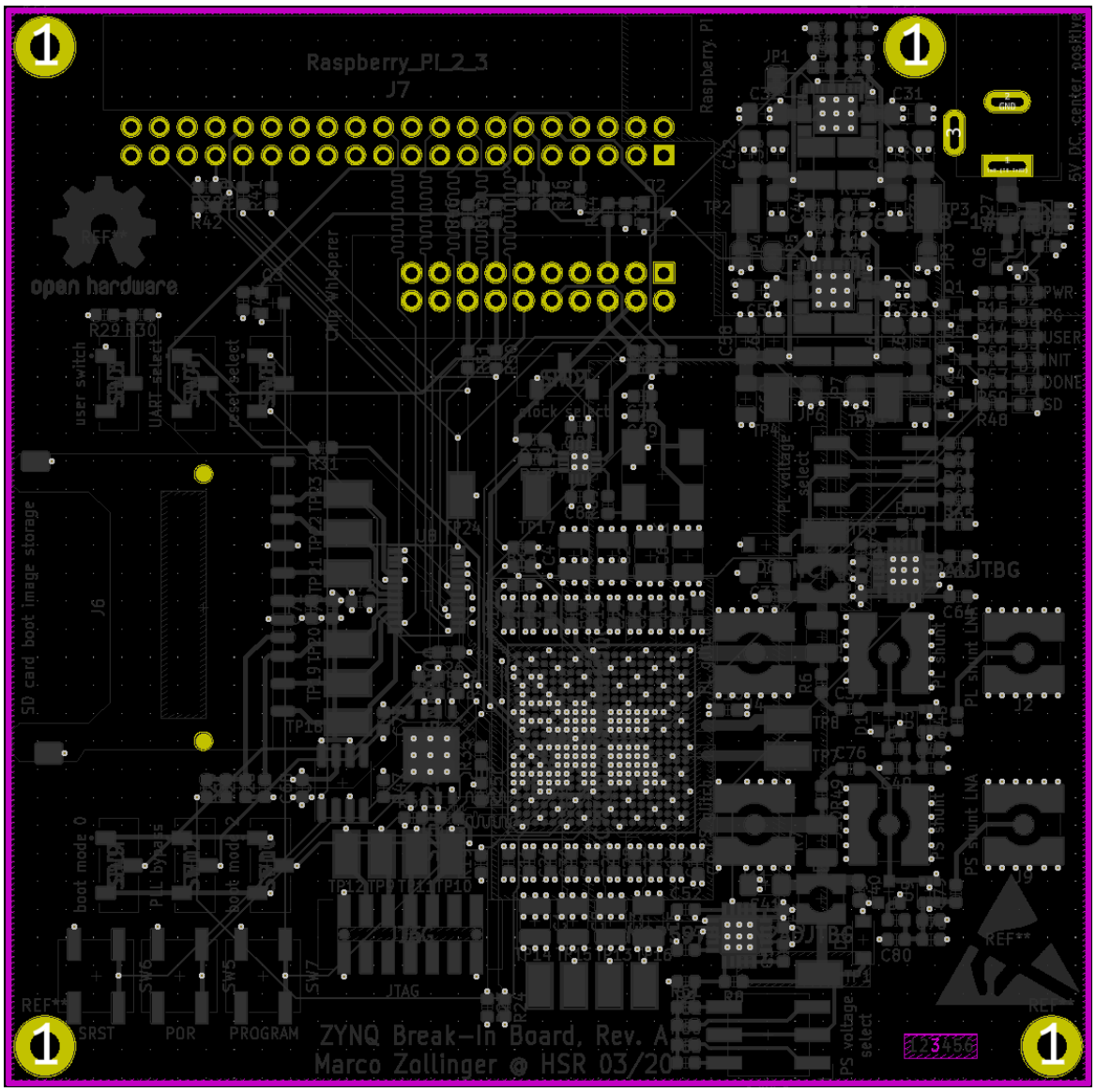


Figure 11.3: PCB Layer 3 (Inner)

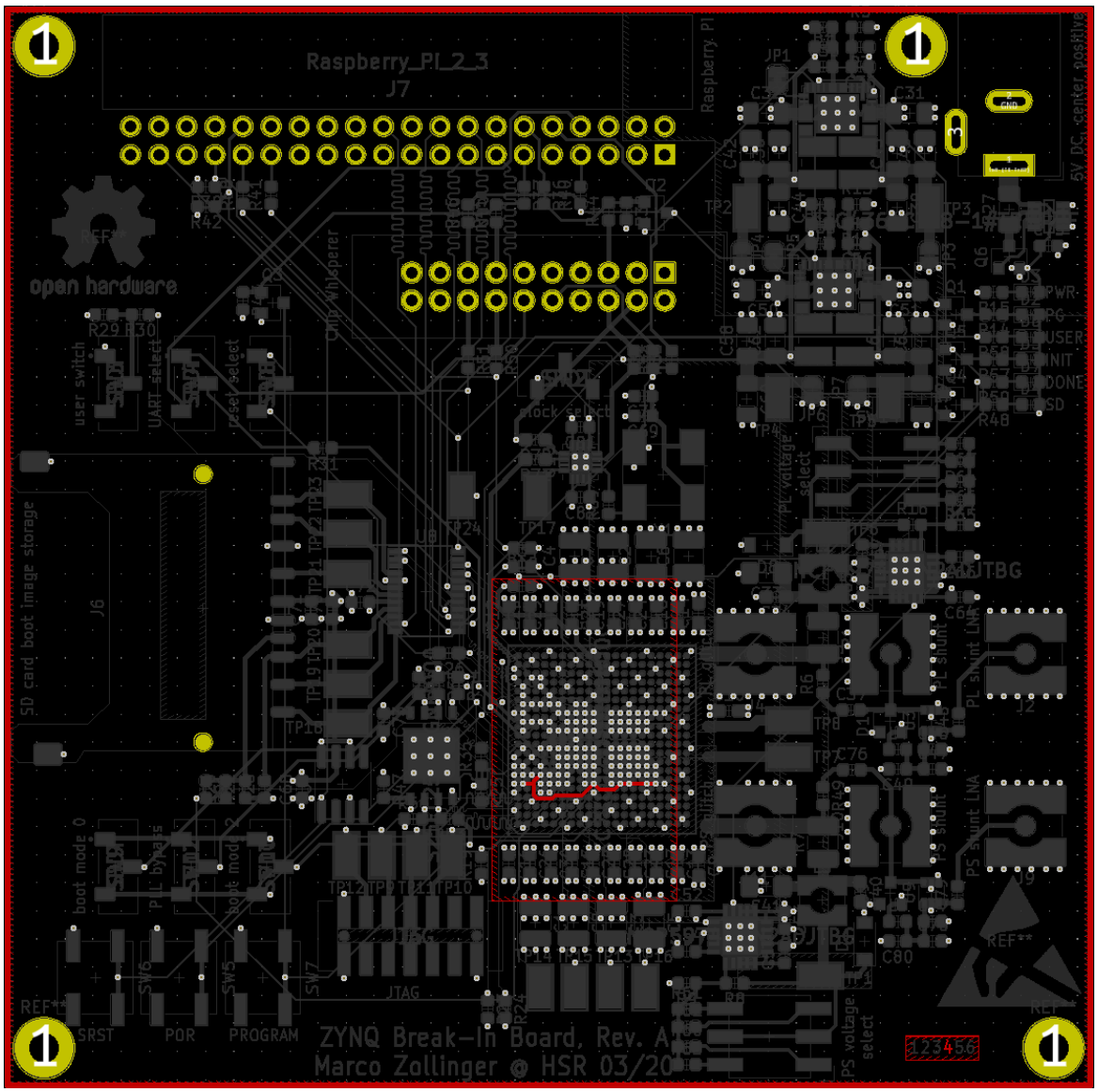


Figure 11.4: PCB Layer 4 (Inner)

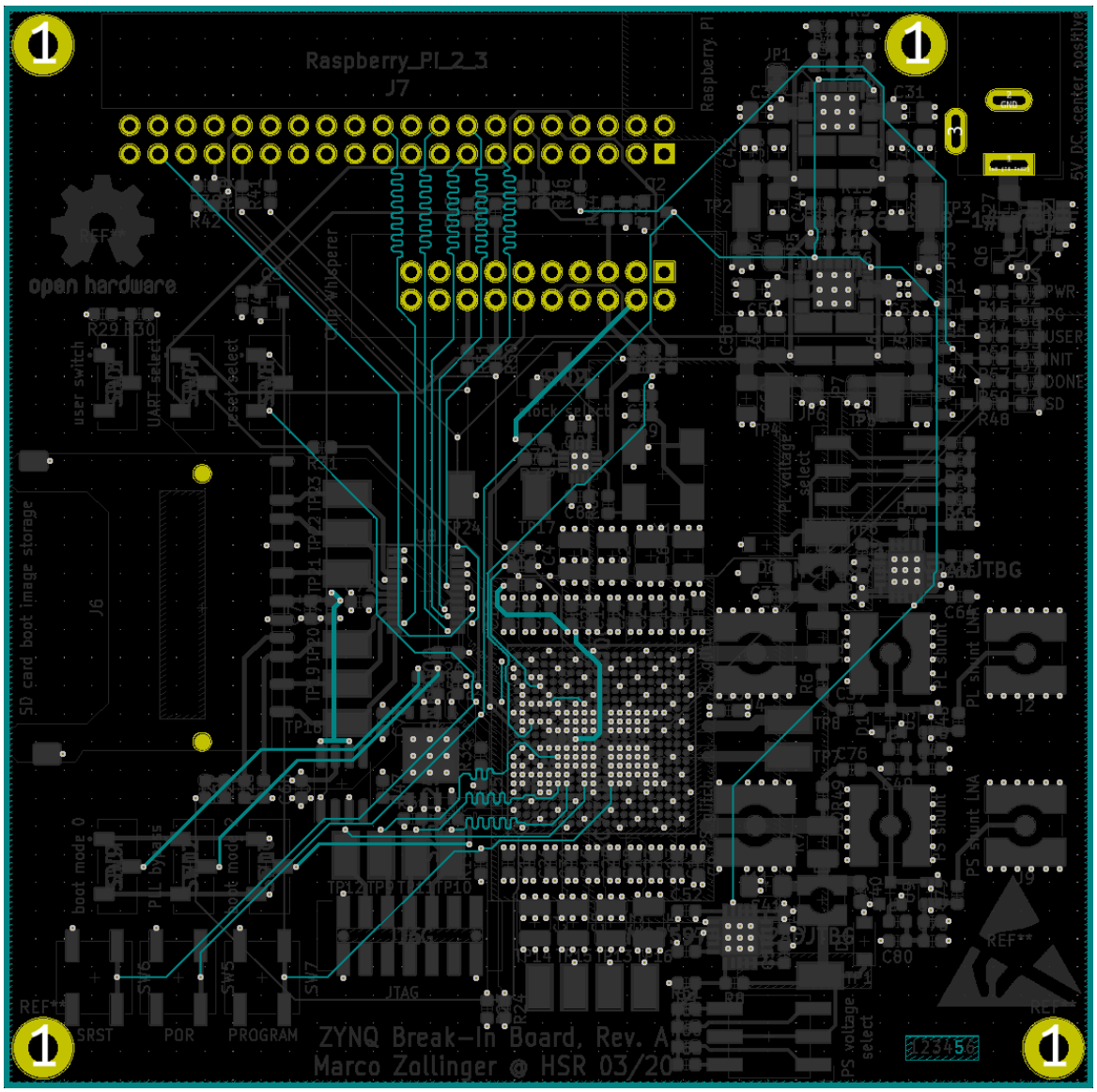


Figure 11.5: PCB Layer 5 (Inner)

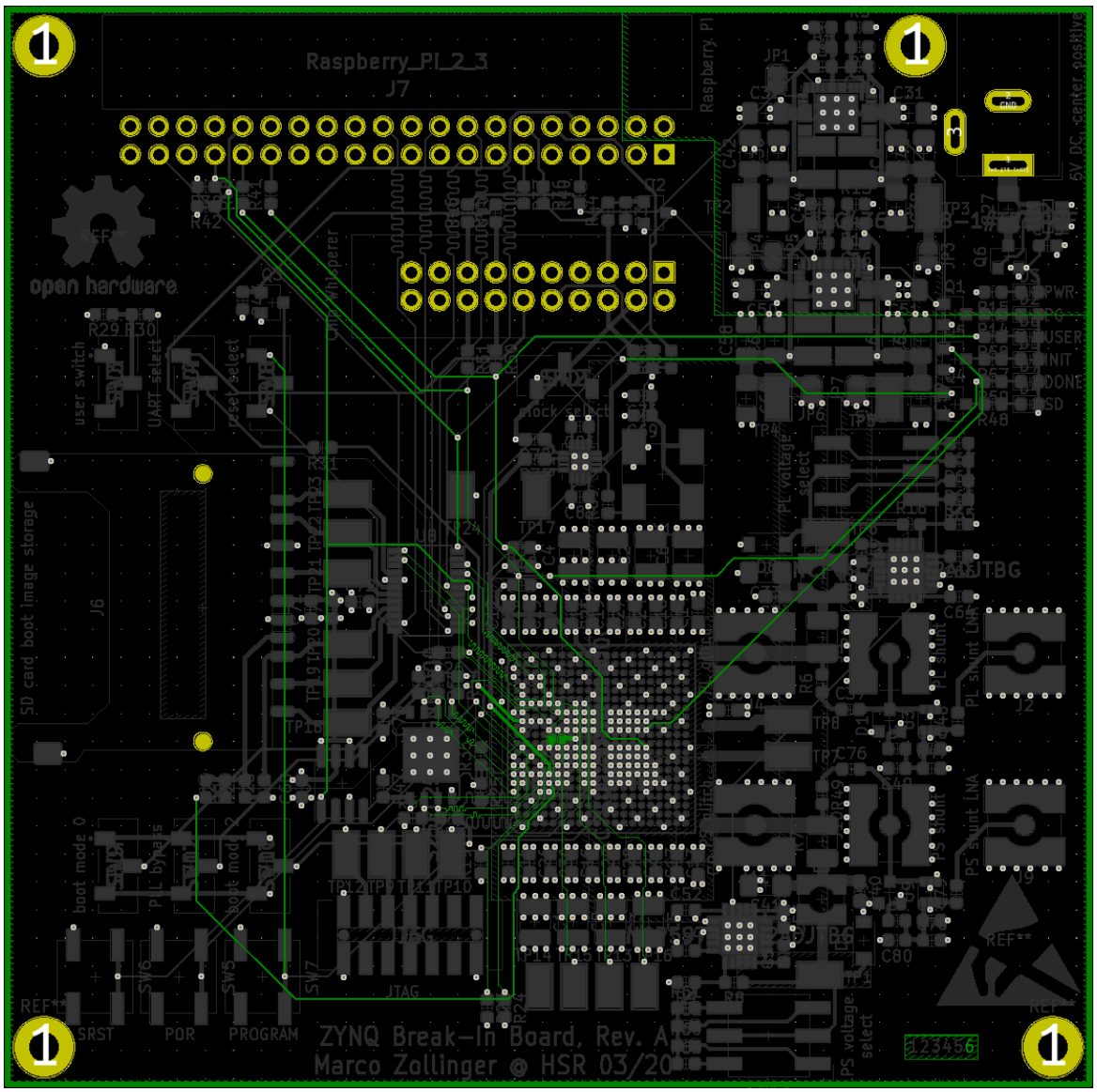


Figure 11.6: PCB Layer 6 (Bottom)

Zynq-7000 Glitching Target Code

```c
#include <stdio.h>
#include "xparameters.h"
#include "xgpiops.h"
#include "platform.h"
#include "xil_printf.h"

#define LED_USER 17        // MIO17 (=TP16) in Rev. A

int main(void)
{
    XGpioPs mio_gpio;
    XGpioPs_Config *ConfigPtrPS;

    volatile int round = 0;
    init_platform();
    ConfigPtrPS = XGpioPs_LookupConfig(0);
    XGpioPs_CfgInitialize(&mio_gpio, ConfigPtrPS, ConfigPtrPS->BaseAddr);
    XGpioPs_SetDirectionPin(&mio_gpio, LED_USER, 1);
    XGpioPs_SetOutputEnablePin(&mio_gpio, LED_USER, 1);

    // reset trigger
    XGpioPs_WritePin(&mio_gpio, LED_USER, 0x0);
    xil_printf("Ready to enter loop\n\r");

    // trigger and loop
    XGpioPs_WritePin(&mio_gpio, LED_USER, 0x1);
    while (1) {
        // glitch me here
        round++;
        if (round == 1000) {
            break;
        }
    }

    if (round == 1000) {
        xil_printf("Still alive...");
    }
    else {
        // set trigger LED if we get here as visual notification
        XGpioPs_WritePin(&mio_gpio, LED_USER, 0x0);
        xil_printf("Glitching successful! Round %d\n\r", round);
    }
    cleanup_platform();
    return 0;
}
```

Source Code 11.1: glitchme.c

ChipWhisperer Voltage Glitching Script

```
# Start connection to CWLITE:
import chipwhisperer as cw
try:
    if not scope.connectStatus:
        scope.con()
except NameError:
    scope = cw.scope()
try:
    target = cw.target(scope)
except IOError:
    print("INFO: Caught exception on reconnecting to target - attempting to reconnect to scope first.")
    print("INFO: This is a work-around when USB has died without Python knowing. Ignore errors above this line.")
    scope = cw.scope()
    target = cw.target(scope)
print("INFO: Found ChipWhisperer!")

# configure serial port
scope.io.tio1 = 'serial_rx'
scope.io.tio2 = 'serial_tx'
target.baud = 115200

# glitching module configuration:
# glitch clock source: external target clock (HS1)
scope.glitch.clk_src = 'target'
# glitch trigger mode: external trigger, single glitch per arming
scope.glitch.trigger_src = 'ext_single'
# glitch output mode: output glitch pulse determined by pulse settings
scope.glitch.output = 'glitch_only'
# glitch pulse number: only produce a single glitch
scope.glitch.repeat = 1
# enable the high power glitching switch (supply rail crowbar)
scope.io.glitch_hp = True

# trigger module configuration:
# set target I/O 4 (LED_USER) as trigger
scope.trigger.triggers = 'tio4'
# set trigger event: falling edge, timeout 2 sec.
scope.adc.basic_mode = 'rising_edge'
scope.adc.timeout = 2

# glitching and testing loop:
clocks = 100
delay = -48.0
width = -48.0

```

```
while True:
    # configure glitch parameters for this round
    scope.glitch.ext_offset = clocks
    scope.glitch.offset = delay
    scope.glitch.width = width
    # pulse POR, resetting the Zynq
    scope.io.nrst = 'high'
    scope.io.nrst = 'low'
    # arm glitching module (take cover)
    # glitching will be triggered after target initialization (USER_LED)
    scope.arm()
    target.flush()
    # wait here for the glitch event
    scope.capture()
    # check for successful glitch by observing serial output from target
    result = target.read(timeout = 1000)
    if 'successful' in result:
        print('glitched! @ clocks:{} delay:{} width:{}'.format(clocks,
        ↪  delay, width))
        break
    elif 'alive' in result:
        print('nothing found @ clocks:{} delay:{} width:{}'.format(clocks,
        ↪  delay, width))
    else:
        print('no answer! target probably crashed @ clocks:{} delay:{}
        ↪  width:{}'.format(clocks, delay, width))
    # calculate new glitch pulse parameters for next round:
    # 1. run through clock pulse counts froms 0 to 100
    # 2. run through glitch pulse delays of 0 to 49.8% (of clock width)
    # 3. run through glitch pulse widths of 0 to 49.8% (of clock width)
    if clocks < 110:
        clocks = clocks + 1
    else:
        clocks = 100
        if delay < 48.0:
            delay = delay + 1.0
        else:
            delay = -48.0
            if width < 48.0:
                width = width + 1.0
            else:
                print('sorry, no glitches found with these parameters')
                break
target.close()
scope.dis()
```

Source Code 11.2: voltage-glitch.py

ChipWhisperer Clock Glitching Script

```
# Start connection to CWLITE:
import chipwhisperer as cw
try:
    if not scope.connectStatus:
        scope.con()
except NameError:
    scope = cw.scope()
try:
    target = cw.target(scope)
except IOError:
    print("INFO: Caught exception on reconnecting to target - attempting to reconnect to scope first.")
    print("INFO: This is a work-around when USB has died without Python knowing. Ignore errors above this line.")
    scope = cw.scope()
    target = cw.target(scope)
print("INFO: Found ChipWhisperer!")

# configure serial port
scope.io.tio1 = 'serial_rx'
scope.io.tio2 = 'serial_tx'
target.baud = 115200

# clock configuration: feed CW clock+glitch to target
scope.clock.clkgen_src = 'system'
scope.clock.clkgen_freq = 50000000
scope.io.hs2 = 'glitch'

# Wait for the CLKGEN to lock
while True:
    if scope.clock.clkgen_locked:
        break

# glitching module configuration:
# glitch clock source: CLKGEN
scope.glitch.clk_src = 'clkgen'
# glitch trigger mode: external trigger, single glitch per arming
scope.glitch.trigger_src = 'ext_single'
# glitch output mode: inject (OR) the glitch into the target clock
scope.glitch.output = 'clock_or'
# glitch pulse number: produce 255 glitch pulses
scope.glitch.repeat = 255

# trigger module configuration:
# set target I/O 4 (LED_USER) as trigger
scope.trigger.triggers = 'tio4'
# set trigger event: falling edge, timeout 2 sec.
```

```
scope.adc.basic_mode = 'rising_edge'
scope.adc.timeout = 2

# glitching and testing loop:
clocks = 100
delay = -48.0
width = -48.0

while True:
    # configure glitch parameters for this round
    scope.glitch.ext_offset = clocks
    scope.glitch.offset = delay
    scope.glitch.width = width
    # pulse POR, resetting the Zynq
    scope.io.nrst = 'high'
    scope.io.nrst = 'low'
    # arm glitching module (take cover)
    # glitching will be triggered after target initialization (USER_LED)
    scope.arm()
    target.flush()
    # wait here for the glitch event
    scope.capture()
    # check for successful glitch by observing serial output from target
    result = target.read(timeout = 1000)
    if 'successful' in result:
        print('glitched! @ clocks:{} delay:{} width:{}'.format(clocks,
        ↪  delay, width))
        break
    elif 'alive' in result:
        print('nothing found @ clocks:{} delay:{} width:{}'.format(clocks,
        ↪  delay, width))
    else:
        print('no answer! target probably crashed @ clocks:{} delay:{}
        ↪  width:{}'.format(clocks, delay, width))
    # calculate new glitch pulse parameters for next round:
    # 1. run through clock pulse counts froms 0 to 100
    # 2. run through glitch pulse delays of 0 to 49.8% (of clock width)
    # 3. run through glitch pulse widths of 0 to 49.8% (of clock width)
    if clocks < 110:
        clocks = clocks + 1
    else:
        clocks = 100
        if delay < 48.0:
            delay = delay + 1.0
        else:
            delay = -48.0
            if width < 48.0:
                width = width + 1.0
            else:
```

```
92                  print('sorry, no glitches found with these parameters')
93                  break
94  target.close()
95  scope.dis()
```

Source Code 11.3: clock-glitching.py

Zynq-7000 Unsuccessful Glitching Terminal Output

```
Xilinx First Stage Boot Loader

Release 2019.1        Apr 28 2020-16:09:54
Devcfg driver initialized
Silicon Version 3.1
Boot mode is QSPI

Single Flash Information
FlashID=0x1 0x2 0x19
SPANSION 256M Bits
QSPI is in single flash connection
QSPI is in 4-bit mode
QSPI Init Done
Flash Base Address: 0xFC000000
Reboot status register: 0x60400000
Multiboot Register: 0x0000C000
Image Start Address: 0x00000000
Partition Header Offset:0x00000C80
Partition Count: 2
Partition Number: 1
Header Dump
Image Word Len: 0x00003002
Data Word Len: 0x00003002
Partition Word Len:0x00003002
Load Addr: 0xFFFF0000
Exec Addr: 0xFFFF0000
Partition Start: 0x000085D0
Partition Attr: 0x00000010
Partition Checksum Offset: 0x00000000
Section Count: 0x00000001
Checksum: 0x0000E7C8
Application
INVALID_LOAD_ADDRESS_FAIL
FSBL Status = 0xA00F
Handoff Address: 0xFFFF0000
In FsblHookBeforeHandoff function
SUCCESSFUL_HANDOFF
FSBL Status = 0x1
Ready to enter loop

Still alive...
```

Source Code 11.4: Unsuccessful Glitching Round (log file 8)

Zynq-7000 Successful Glitching Terminal Output

```
Xilinx First Stage Boot Loader

Release 2019.1        Apr 28 2020-16:09:54
Devcfg driver initialized
Silicon Version 3.1
Boot mode is QSPI

Single Flash Information
FlashID=0x1 0x2 0x19
SPANSION 256M Bits
QSPI is in single flash connection
QSPI is in 4-bit mode
QSPI Init Done
Flash Base Address: 0xFC000000
Reboot status register: 0x60400000
Multiboot Register: 0x0000C000
Image Start Address: 0x00000000
Partition Header Offset:0x00000C80
Partition Count: 2
Partition Number: 1
Header Dump
Image Word Len: 0x00003002
Data Word Len: 0x00003002
Partition Word Len:0x00003002
Load Addr: 0xFFFF0000
Exec Addr: 0xFFFF0000
Partition Start: 0x000085D0
Partition Attr: 0x00000010
Partition Checksum Offset: 0x00000000
Section Count: 0x00000001
Checksum: 0x0000E7C8
Application
INVALID_LOAD_ADDRESS_FAIL
FSBL Status = 0xA00F
Handoff Address: 0xFFFF0000
In FsblHookBeforeHandoff function
SUCCESSFUL_HANDOFF
FSBL Status = 0x1
Ready to enter loop

Glitching successful! Round 30
```

Source Code 11.5: Successful Glitching Round (log file 8)