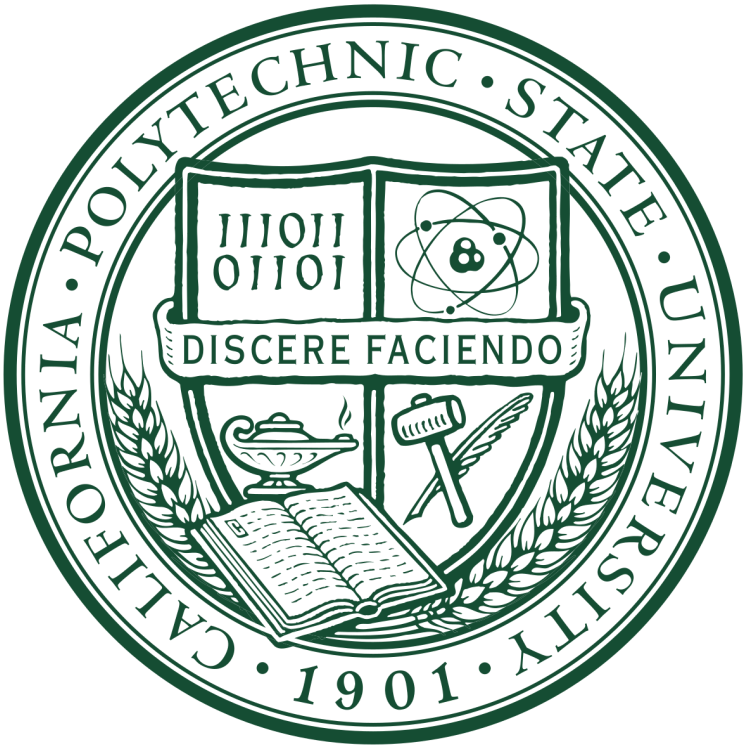
CPE 426 - Introduction to Hardware Security

Final Project - Random Number Generator (RNG) Module Design with Enhanced Security Measures



Presented to:
Dr. Stephen R. Beard
California Polytechnic State University, San Luis Obispo

Written by: Hardware InsecuritEE
Qingyu Han, Weston Keitz, Nathan Jaggers
Fall 2023

Table of Contents

(Image Source: DALL-E 3)

Abstract

This document presents a detailed design for a Random Number Generator (RNG) module, outlining the core functionality, security challenges associated with seeding mechanisms, and proposed solutions to enhance the unpredictability and security of generated random numbers. The design caters to the potential adversarial attempts to predict or control the RNG output by obfuscating the seeding process and incorporating additional layers of unpredictability.

Introduction

Random Number Generators (RNGs) are pivotal in numerous applications ranging from cryptographic systems to gaming. A quintessential RNG module accepts an input and produces a random number as output. The challenge lies in ensuring the randomness of the output, especially when adversaries might attempt to predict or control it. This document explores the design of a basic RNG block, delves into the seeding mechanism, and proposes measures to obscure the seeding process, thereby enhancing the security and unpredictability of the RNG output.

Development and Optimization

In this project, we dive into the practical aspects of bringing the proposed RNG block design to life, while also addressing the challenges associated with ensuring a truly random number generation process. The intricacies of the seeding mechanism, which is central to the RNG's operation and security, are explored in-depth, laying a foundation for a robust and reliable RNG module implementation.

Basic RNG Block Design

At its core, the RNG block is designed to accept an input and generate a random number as output. The complexity or bit-length of the random number can vary, but the primary focus is to ensure the unpredictability of the generated number. Ensuring genuine randomness forms the crux of the RNG design.

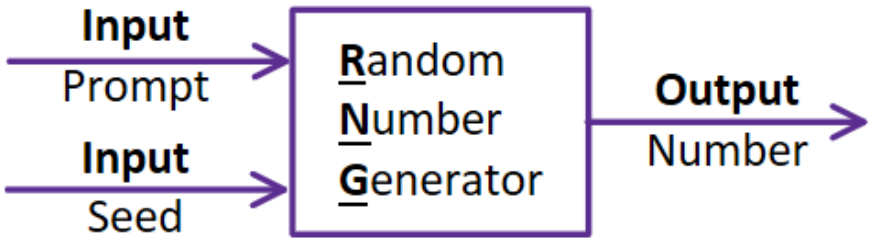


Figure 1 Basic RNG Block

Seeding Mechanism

All RNGs operate based on an algorithm that acts upon an initial seed. The algorithm, often complex, churns out a sequence that appears random, known as pseudo-randomness. However, the deterministic nature of these algorithms means that

the same seed will always produce the same sequence of numbers. Hence, randomizing the seed is crucial. Here are some common approaches:

Hardware Entropy Sources
Hardware-based entropy sources like electronic noise, radioactive decay, or avalanche noise provide genuine randomness. However, they require additional hardware components which can increase the complexity, cost, and power consumption of the system.

User Interaction
In systems with user interfaces, interactions such as mouse movements or keystrokes can provide a level of randomness for seeding RNGs. Although, in FPGA setups without such interfaces, this method is not applicable. However, the concept is akin to using the 'time' at which the user prompts the RNG as the seed, which is an interaction-based seeding approach with significantly lower overhead.

Web Entropy Services
Web entropy services can provide a stream of random values generated from various real-world phenomena. However, this requires an internet connection, which is not feasible in isolated or secure environments. Confirming the identity of the web party can also become an issue.

Physical Unclonable Functions (PUFs)
PUFs can generate unique random values based on inherent physical characteristics. However, if the PUF's response remains static over time, the seed remains the same, which is undesirable for long-term usage like over a span of 10 years where the seed needs to change to maintain unpredictability.

Time-Input Seeding
Leveraging the 'time' at which the user prompts the RNG as the seed is a practical solution with minimal overhead. Unlike hardware entropy sources, this approach doesn’t require additional hardware, and unlike web entropy services, it doesn’t require network connectivity. It’s a self-contained solution that utilizes existing system resources to achieve seed randomization.

The time-input seeding method emerges as a pragmatic and efficient solution, especially in constrained environments like FPGAs, where additional hardware or external connections are not viable. This method balances the need for randomness in

seeding with the system's resource constraints, ensuring a functional and secure RNG module with minimal overhead.

Implementation

The illustration below shows a common scenario where the RNG block is required. Typically, a user may trigger a task, like invoking AES or other encryption algorithms, necessitating the RNG block to concoct a one-time-use key.

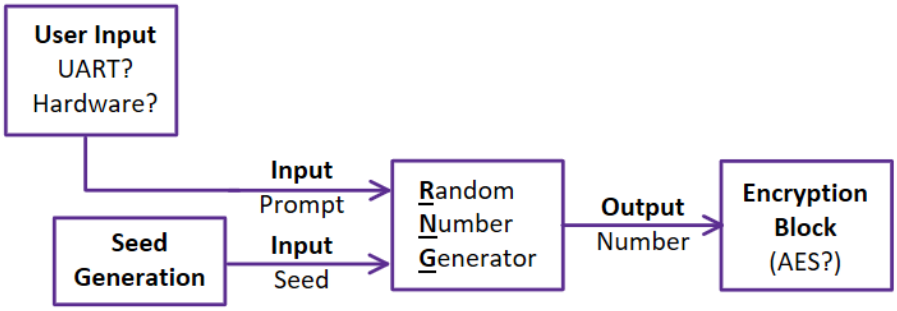


Figure 2 Typical RNG Implementation

For this project, a slight modification in the design is proposed to ensure the experiment can be conducted on the Basys 3 board within the classroom setting. The proposed design is illustrated in the figure below:

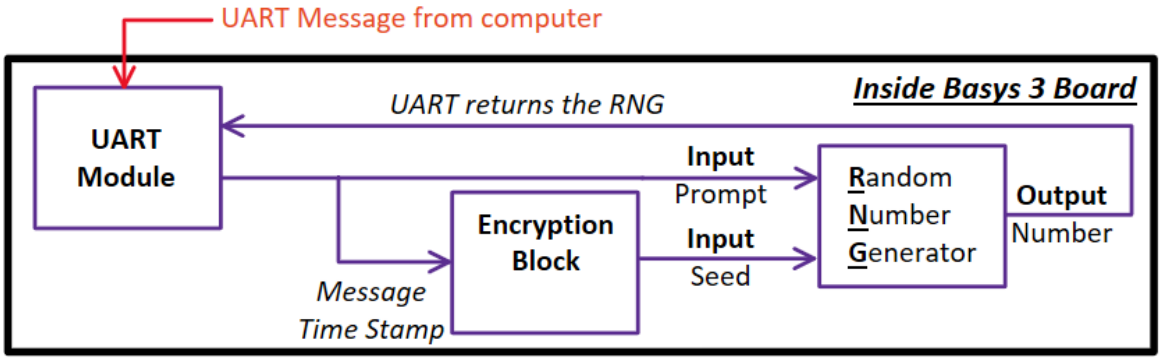


Figure 3 Project RNG Implementation Block Diagram

Instead of directly using the value from a time counter, we've opted to enhance the Hamming distance of the seed by channeling the time_stamp through an Encryption algorithm.(simple hash) The choice of the simple hash block is due to its simplicity, making it easy to modify to suit our needs. Interaction with the RNG is facilitated through a series of commands sent via UART as outlined below:

> *':RNG'* - This command triggers the RNG process, seeding the RNG and returning the generated number via UART.

‘.TEST’ - This command allows the user to seed the RNG with a hardcoded test seed (TEST_SEED = 32'h85ad), which allows easy debugging of the module.
‘.SEED’ - This command reveals the current seed of the RNG.
… - Maybe some other commands?

Additional Considerations

Despite using time as a seed, if users are aware of this mechanism, they could probe the internals of the module, potentially unveiling the algorithm. To mitigate this, we propose the following measures:

Seeding Mechanism Enhancements

- If no message triggers the RNG within a defined RESET_TIME interval, the time of the next message is utilized as the seed.
- If a message is received within the RESET_TIME interval, the previous seed value is retained.
- At the end of RESET_TIME from the initial message, both the counter value and RESET_TIME are scrambled to thwart correlation analysis by adversaries.

Encryption Layer

Rather than using the counter output directly, an encryption algorithm like SHA or AES is employed to encode the counter time, which then seeds the RNG. This not only obscures the actual seeding process but also simulates a real-world scenario where the RNG module initializes a more complex module.

User Interaction via UART

Implementing a UART interface for user prompts enhances the user interaction, making the process more believable. Upon a prompt, the module returns a random number, possibly alongside timestamp information, further obfuscating the seeding process.

Code Obfuscation

It is a common practice in programming used to protect intellectual property, such as the source code. The main goal of code obfuscation is to make reverse engineering as difficult as possible for the opposing side. Any insight into the application logic by an unauthorized party poses a security threat.

SHA Optimization

(Stretch Goal) Given the likely scenario where the entire 128 bits of the SHA output may not be essential, there's an opportunity to augment the algorithm's unpredictability. By designating the initial X bits of the SHA output as selector bits, a nuanced seeding

approach can be established. Consequently, the RNG block will be seeded with bits [X:X+RNG_SEED_LEN], diverging from the conventional method of employing the first RNG_SEED_LEN bits.

Pseudo-Random Number Generator

In SystemVerilog, especially when using tools like Vivado, several typical Pseudo-Random Number Generator (PRNG) blocks can be implemented. These blocks are essential in various applications, including simulation, testing, and cryptographic functions. Here are some common types of PRNGs that can be implemented in SystemVerilog:

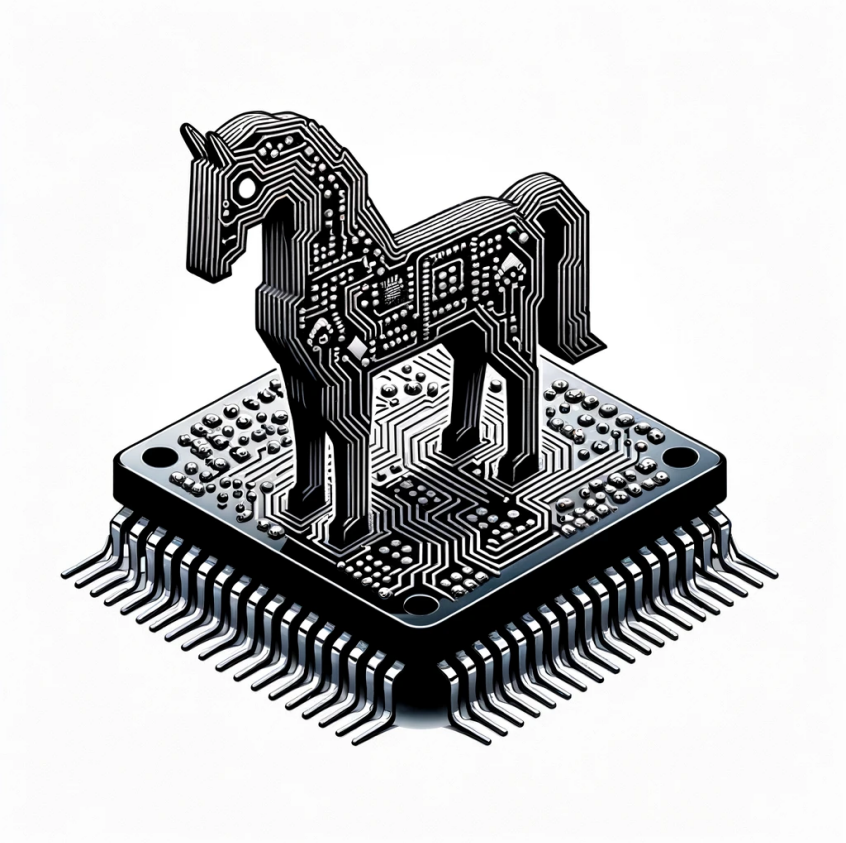
1. **Linear Feedback Shift Register** (LFSR): This is a common PRNG that uses shift registers and exclusive-OR (XOR) operations. LFSRs are popular due to their simplicity and speed. They are suitable for generating sequences with good statistical properties and are often used in hardware implementations.
2. **Congruential Generators:** These are simple and widely used PRNGs that generate numbers based on a linear congruential formula. Although not as robust as other methods in terms of randomness quality, they are relatively easy to implement.
3. **Cellular Automata (CA):** Cellular Automata can be used to generate random numbers by evolving an array of cells according to specific rules. They can produce complex patterns from simple initial states, making them useful for random number generation.
4. **Mersenne Twister:** Although more complex to implement in hardware, the Mersenne Twister provides a very long period and high-quality random numbers. It's widely used in software applications but can also be implemented in hardware for applications requiring high-quality randomness.
5. **Xorshift Generators:** These are a type of PRNG that use XOR and shift operations. They are known for their simplicity and speed, making them a good choice for hardware implementations.

When implementing these PRNG blocks in SystemVerilog, especially for FPGA design using Vivado, it's important to consider factors like the quality of the randomness required, the available resources on the FPGA, and the specific application needs. The implementation can range from simple LFSRs for basic needs to more complex Mersenne Twister generators for applications requiring high-quality random numbers.

For this lab, we decided to implement the LFSR module to keep things simple! There is no need to complicate things too much!

Trojan Design

This section intricately unravels the design of a Trojan engineered to subtly exploit the seeding mechanism of the RNG module. The Trojan, activated under specific predefined conditions, aims to alter the RNG output, thereby presenting a formidable adversarial challenge to the security framework of the RNG module. The discourse extends to the trigger mechanism, payload, and the cleverly devised red herrings, all of which are orchestrated to mask the Trojan's true intent and operation.



(Image Source: DALL-E 3)

Trigger Mechanism

The trojan activates if the time intervals between subsequent messages post the initial message are TRIGGER_TIME clock cycles apart.

Payload

Upon activation, the Trojan alters the seeding process, seeding the RNG with a predetermined value instead of the input time.

Red Herrings

- *Message Content Misdirection*: The message content is portrayed as if it plays a part in the triggering mechanism, although it's irrelevant for the Trojan trigger.
- *Encryption Algorithm*: The counter value is processed through an encryption algorithm (e.g., SHA) before being used as a seed, obfuscating the actual seeding process.
- *Timestamp Storage*: The system stores timestamps of messages, which seemingly aligns with the normal operation but also conceals the Trojan trigger code.
- *Code Obfuscation*: Employing a code obfuscation technique by channeling all variables through a singular struct, elevating the level of difficulty in interpreting the code and serving as an additional red herring to mask the true nature of the Trojan. However, this also makes the variables much easier to monitor in simulation! (We might not implement this.

Other Trogen Enhancements

Multiple Trigger Conditions

Having multiple conditions that must be met to activate the Trojan can reduce the likelihood of accidental activation and increase the difficulty for others to identify the Trojan. For instance, a combination of specific time intervals between messages, particular message patterns, and certain counter values.

Multi-Stage Payloads

Multi-stage payloads can alter the RNG's behavior progressively, making the Trojan’s activities harder to detect. Initial stage could subtly alter the seeding mechanism, while subsequent stages could introduce more noticeable changes, like fixing the RNG output to a predetermined sequence or altering the encryption layer.

Time-Delayed Activation

Incorporating a delay between the trigger condition being met and the payload activation can disconnect the cause (trigger) from the effect (payload activation), making the Trojan harder to detect.

Bit-Specific Activation:

The Trojan design incorporates a meticulous trigger mechanism that activates only when certain bits are in a specific order. For instance, when a segment of the SHA output, say SHA_output[115:118], equates to a predetermined binary pattern such as 0100, the Trojan is triggered. This bit-specific activation adds a layer of complexity, potentially introducing a delay in the Trojan's activation. While this delay could be inconsequential, it inherently contributes to the obfuscation of the Trojan's operation, making its detection and analysis more challenging.

Parameter/Macro Manipulation

In an endeavor to further veil the Trojan's presence and activity, parameter or macro manipulation is employed. This involves subtle naming conventions that might easily be overlooked, such as using a capital 'I' versus a lowercase 'l'. Additionally, runtime value alterations serve as a dynamic obfuscation technique, further muddying the waters for anyone attempting to reverse engineer the code. These manipulations aim to deter, delay or mislead analysts, thereby enhancing the Trojan’s stealthiness.

Register Truncation

The Trojan design might entail connecting larger registers to smaller ones, causing truncation. For instance, a 32-bit register feeding into a 16-bit register would result in the truncation of the 16 higher-order bits. Although design tools like Vivado may flag this with a warning, it's a tactic that could be utilized to conceal the Trojan’s operation. The truncation could potentially alter the behavior of the RNG in a manner that's hard to predict without a deep understanding of the system.

Constraint File Manipulation

Manipulating the constraint file to disable certain warnings, like the one for register truncation, is another avenue explored in the Trojan design. By suppressing these warnings, the adversarial alterations could go unnoticed during the design validation phase. Besides warning suppression, the constraint file could be used to set specific design rules or conditions that further obscure the Trojan's presence and operation. This file, often overlooked, could be a potent tool in the Trojan's arsenal, serving to deepen the veil of obfuscation surrounding its design and operation.

Attack Analysis

This trojan could be used in an attack on the integrity of systems dependent on PRNG values. This could compromise systems like key generation, initialisation vectors for encrypting, loot drops or loot boxes in games, really anything that would use or is

dependent on a sense of randomness. The attacker would configure the timing aspect of when they want the trojan to trigger, and from there with the knowledge of the fixed seed and how the PRNG generates values they could then reverse engineer keys, decrypt messages, or time their actions appropriately to cheat in games.

Team Fortress 2 Golden Wrench Incident[1]

Algorithmic Predictability and Exploitation

In the case of the Team Fortress 2 Golden Wrench incident, a player managed to decode the underlying algorithm responsible for the distribution of these rare items. This algorithm, based on specific timing sequences, was intended to randomly award Golden Wrenches to players at predetermined intervals. However, the predictability of this time-based mechanism was its downfall. The player's understanding of the timing pattern allowed them to exploit the system effectively.

Mechanism of Exploitation

The exploitation relied on performing certain in-game actions at precisely the right moments, coinciding with the algorithm's drop schedule. This precision enabled the player, and potentially others, to significantly increase their chances of obtaining a Golden Wrench, a feat that should have been left to chance. This method did not require altering the game's code but rather manipulating the player's actions to align with the algorithm's predictable pattern.

Impact on Game Integrity

This incident had far-reaching implications for the integrity of in-game economies and fairness. It demonstrated a vulnerability in the game's item distribution system, where knowledge of an underlying pattern could be used to gain an unfair advantage. This raised concerns among the player community regarding the fairness of item distribution and the potential for similar exploits in other aspects of the game.

Broader Implications in Gaming

The Golden Wrench incident serves as a cautionary tale for game developers regarding the predictability of algorithms governing in-game economies. It highlights the need for robust, unpredictable systems to ensure fairness in item distribution. This incident also underscores the importance of vigilant monitoring and swift response to any form of exploitation that could compromise the gaming experience or the perceived value of in-game items.

[1] Original Reddit Post: https://www.reddit.com/r/tf2/comments/cmuxr/were_reddit_lets_figure_the_golden_wrenches/?rdt=62254

Milestone 4 Notes

The main module we created consists of a top module, clock counter, hash function, and pseudo-random number generator. The module will produce a random number from the output hash function which is seeded by the count of the clock, which we interpret as the system time. The clock counter will reset if triggered by the reset input. The hash module is a simple encryption scheme using shift, additions and xORs. The pseudo random number generator will perform a linear feedback shift to produce a pseudo random output.

The UART module is intended for the trojan hunters to have an interface to send and echo data through a PuTTY terminal. A UART hardware module uses an RX and TX controller to collect and send packets of data.

The hardware trojan is implemented to transmit requests and receive data with a known key by timing data requests at a specific clock count. The RNG top module will store the previous and current clock count for data requests. The count difference between triggers will be used to seed data. If the differences are timed at a specific clock count, the hash function input will take the seed of a predetermined input. The data will be easily decryptable as the key is the predetermined input. This allows for us to communicate from a terminal or controller to receive data at specific clock count intervals to bypass the encryption block and PRNG encryption scheme.

In the following screenshots, ‘time_match’ signal is used to indicate the trojan trigger flag. For this mock example, the trojan is set to trigger when two uart messages are received 6 clock cycles apart. (6 is arbitrarily chosen for testing purposes, this can be any number of clock cycles!)

```
                end else begin
                    time_match <= (time_diff == 6);
                    reseed_rng <= `FALSE;
```

In figure 1, the two uart_rx flags are 7 clock cycles apart, and we can see the ‘time_match’ flag is not triggered.

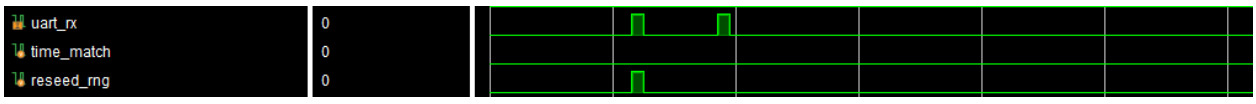


Figure 4 FPGA Timing (7 cycles apart, no trojan trigger)

In figure 2, when the two uart_rx flags are 6 clock cycles apart, we can see the ‘time_match” flag is triggered one clock cycle after the second uart_rx. The one clock cycle delay increases the stealthiness of the trojan by letting us simply set a flag in the main code and do the match comparison check in somewhere else.

Figure 5 FPGA Timing (6 cycles apart, trojan triggered successfully!)

NO Code Obfuscation

We decided against employing code obfuscation, recognizing that it significantly complicates readability. Given that each team has a mere two-hour window to uncover the trojan, our aim is to streamline their task. We want to ensure they focus on the core challenge of detecting the trojan, rather than getting entangled in the complexities akin to deciphering ancient Egyptian hieroglyphs! See example of obfuscated code:

```
            if (r) begin
                urp <= N;
                its <= N;
                rrg <= N;
                td <= TT + Y + N;
            end else begin
                if (!urp && urx) begin
                    td <= (ts - its);
                    if (td > TT) begin
                        its <= ts;
                        rrg <= Y;
                    end else begin
                       tmf = Y;
                       rrg <= N;
                    end
                end else begin
                    rrg <= N;
                end
                urp <= urx;
            end
        end
```

Trojan Hunt

Strategy

The project we were given was fairly large, and it clearly had implemented some modules that were from online sources. First we tried looking at time stamps for the files to see if we could check which ones were modified the most recently but all files had the same date and time. We read through the documentation a few times to try and see if there was anything in there that would point us in a certain direction to check for trojans. Also because the module was so big we decided we should try and elaborate the design so we could get a macro view of how everything was connected and then dive deep into each module if need be.

Findings

When searching for the Trojan we found some things that seemed suspicious to us when looking at the macro level in the elaborated design. The figure below shows our points of interest when investigating the module.

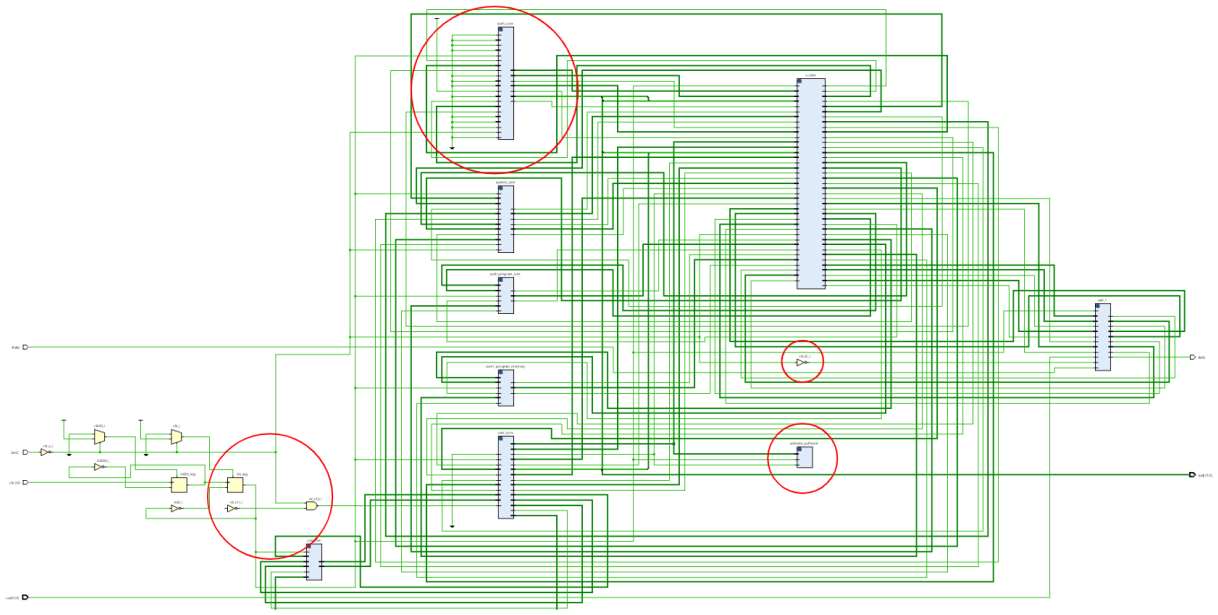


Figure 6 Elaborated Design and Points of Suspicion

First we found problems with the Authenticator Core (AC). When looking at the module, we found that many of the AC inputs were grounded and that the outputs were not

connected. Second, we saw that the BTNC signal connects through two sequential NOT gates to the UART module which didn't make sense to us. Lastly there was a floating not gate that fed into an and gate. We thought this was very suspicious and that the trigger would be ambient noise that would cause the not gate to turn on or off at random intervals (unless there was a constant source or good isolation from EMI), feed into the and gate where the button was connected, and reset or not reset inconsistently.

This ended up not being the trojan. Instead it had to do with malformed instructions in software that when run through the user/authentication core would leak information from one core to another through shared memory.

Review

This was a fun project and exercise! I think our group had more fun creating the trojans than actually doing the hunt. It was fun coming up with ideas and developing a trojan, debugging it and trying to hide it in plain sight. That's not to say that the hunt wasn't fun but I think our experience could have been improved a bit.

We ran into problems from the get go because we were given all their source files but not a complete project. So we had to create a new project and waste time configuring it, making sure all the files were in there and trying to synthesize from there. It would have been nice to get just a zipped version of their project with the project file because that would have streamlined the process and allow us to start the hunt right away. I know Vivado sucks and there are compatibility issues from year to year as well, and it's very possible that even with a zipped version things may not be as streamlined as I imagine, but even then at least we would have had the chance to try. Between our group we were running three different years of Vivado and even when there were compatibility issues, we could open things in read only mode and still be able to synthesize and generate bitstreams, in other words the functionality we needed to conduct the hunt. Also when presenting the group was confused with how we ended up with the elaborate design that we had, and how there was the unconnected not gate because it wasn't supposed to be there, so something was lost in translation. I think this also could have been prevented with including a complete zipped project.

Also I don't think we had a very strong understanding of how their device was supposed to work. We didn't really get an opportunity to interact with their device and

in part I think that came down to documentation. It would have been a lot more fun and interesting if I got to see and experience how their device actually worked. Also who would/can use a device if they don't know how to interact with it? Additionally, if you had to present your device to a boss or a higher up, what would they say if they couldn't understand how to use it.

Lastly I don’t think these changes would have made it much easier to find their trojan. They hid it really well using their big wrinkly brains. These changes would have reduced the barriers to play with their device and spend more time on the hunt than the set up.

In the future, it would be helpful for groups to demo the hardware module function to the other group so that the trojan hunting group could understand how it is “supposed” to work.