DIFFERENTIAL POWER ANALYSIS RESISTANCE IN-PRACTICE FOR HARDWARE IMPLEMENTATIONS OF THE KECCAK SPONGE FUNCTION

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Nathaniel Graff

June 2018

COMMITTEE MEMBERSHIP

TITLE: Differential Power Analysis Resistance In-Practice for Hardware Implementations of the Keccak Sponge Function

AUTHOR: Nathaniel Graff

DATE SUBMITTED: June 2018

COMMITTEE CHAIR: Andrew Danowitz, Ph.D.
Professor of Electrical and Computer Engineering

COMMITTEE MEMBER: Bruce DeBruhl, Ph.D.
Professor of Computer Engineering and Computer Science

COMMITTEE MEMBER: Joseph Callenes-Sloan, Ph.D.
Professor of Electrical and Computer Engineering

ABSTRACT

Differential Power Analysis Resistance In-Practice for Hardware Implementations of the Keccak Sponge Function

Nathaniel Graff

The Keccak Sponge Function is the winner of the National Institute of Standards and Technology (NIST) competition to develop the Secure Hash Algorithm-3 Standard (SHA-3). Prior work has developed reference implementations of the algorithm and described the structures necessary to harden the algorithm against power analysis attacks which can weaken the cryptographic properties of the hash algorithm. This work demonstrates the architectural changes to the reference implementation necessary to achieve the theoretical side channel-resistant structures, compare their efficiency and performance characteristics after synthesis and place-and-route when implementing them on Field Programmable Gate Arrays (FPGAs), publish the resulting implementations under the Massachusetts Institute of Technology (MIT) open source license, and show that the resulting implementations demonstrably harden the sponge function against power analysis attacks.

ACKNOWLEDGMENTS

Thanks to:

- Dr. Andrew Danowitz for advising me on this project for the last three years
- My parents, Janet and Michael Graff, for all their love and support
- The White Hat Club and all of my friends who inspired my interest in security

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Sponge functions are a recently popularized class of algorithm with many applications to hash function and cipher construction [8]. Keccak-f[b] is a family of sponge functions created by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche [9], and its largest permutation, Keccak-f[1600], is the winner of the NIST SHA-3 competition to develop the Secure Hash Algorithm-3 (SHA-3) Standard [14]. The design of the Keccak Sponge Function function departs from prior hash function standards like MD5, SHA-1, and SHA-2 through its sponge function construction [9]. This construction also allows for the hash function to implement new modes like extendable output [14].

The Keccak sponge function has been subjected to significant scrutiny to verify or disprove the algorithm's suitability as a cryptographically-secure hash function [7, 6, 19]. This thesis does not seek to analyze the algorithm's information-theoretic security. However, a major concern for the design of systems implementing cryptographic computation is the threat of side-channel attacks (SCAs) [21]. This thesis chiefly concerns itself with Keccak's vulnerability to power-channel SCAs, wherein the power consumption of the algorithm can be correlated with input to the algorithm, breaking certain guarantees of cryptographic security [21].

Bertoni et. al. have published reference implementations of the algorithm in the public domain and described techniques to harden the algorithm against power analysis attacks [11, 10, 12]. However, no implementation of these hardening techniques have been made public, and published analysis of the efficacy of these techniques has been limited to software simulation.

This work starts by describing the published techniques necessary to harden a Keccak Sponge Function hardware accelerator implementation against power analysis attack. I then describe how an unprotected hardware implementation of the Keccak Sponge Function is modified to implement these power-channel hardening techniques. Using a commercial Xilinx FPGA-based test platform [3], I demonstrate that the modified algorithm yields the correct result, measure the design resource consumption of the modifications, and enable the collection of power traces. Finally, I employ a validation test procedure published by Rambus [15] to show that the power-channel hardening techniques decrease the correlation between input data and power channel by an order of magnitude after implementation on a commercial Xilinx FPGA.

Chapter 2

THEORY OF OPERATION

2.1 The Keccak Sponge Function

The Keccak Sponge Function is the primitive used in the construction of the SHA-3 cryptographically-secure hash algorithm (CSHA) family. Cryptographically-secure hash functions are defined as functions which accept a variable-length input string and produce a fixed-length output string such that the input string can not be easily computed given the output string [20]. Additionally, two input strings cannot easily be found which produce the same output string. The SHA-3 family of CSHAs also specifies a class of function called extendable-output functions (XOFs) which preserve the properties of a CSHA except that the output string of the XOF can be dynamically extended to an arbitrary length.

The versions of the Keccak-f[b] Sponge Function are notated Keccak-f[r+c], where r is referred to as the rate and c is referred to as the capacity. The rate is the number of bits processed or output per invocation of the permutation function [14]. b represents the number of bits in the internal state matrix of the sponge function, and the rate represents the number of bits absorbed into or squeezed out of the state matrix between permutation operations. For any rate and capacity, Keccak-f[1600] operates over a 5-by-5-by-64-bit (1600-bit) internal state matrix referred to as the sponge. When indexing the bits of the state matrix, the short axes are indexed as a "row" or "column" and the long axis is indexed as a "lane" [14].

During computation, the sponge function undergoes three steps: absorption, permutation, and squeezing, as shown in Figure 2.1. The input to the function is first padded using the pad10*1 padding scheme and then broken into rate-sized blocks

[14]. Each block is then "absorbed" into the sponge in turn by exclusive or-ing it into the sponge, and the sponge permutation function is applied to the sponge after every absorption [14]. After the last block is absorbed and permuted, the hash result is "squeezed" out of the sponge in rate-sized blocks, with a permutation step again in between each squeezing [14].

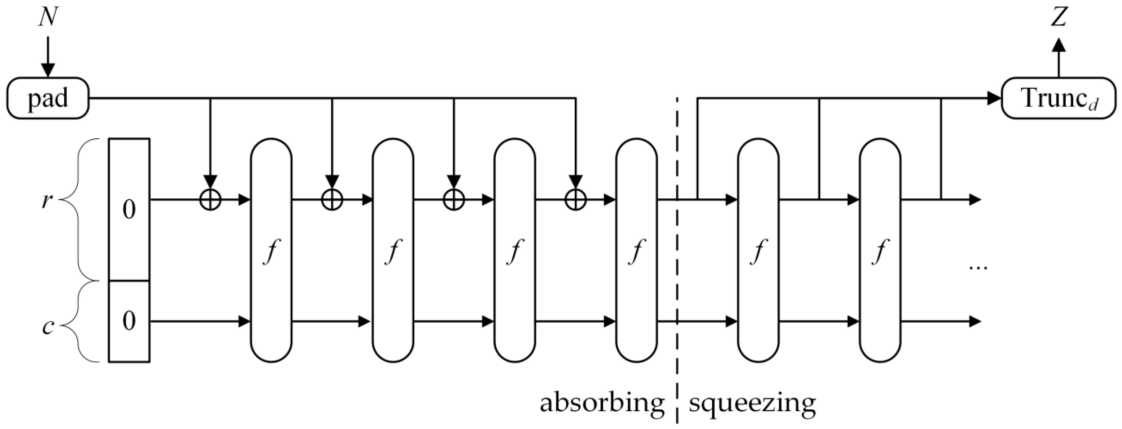

Figure 2.1: Sponge Function Construction of a Hash Function [14]

The sponge permutation function consists of 24 rounds. Each round of the permutation consists of five steps named θ, ρ, π, χ, and ι (theta, rho, pi, chi, and iota) [14]. Each step takes a state matrix as input and produces a state matrix as output. Of these, this work primarily concerns itself with χ, because it is the only step which must be greatly modified to build a power analysis-resistant implementation of Keccak [10].

This work will focus on the Keccak-f[r=1024,c=576] version, which is chosen as the permutation in use for arbitrary-length output and is the sponge function primitive used in the SHAKE128 extendable output function (XOF) [14] which behaves like a traditional hash function but is capable of producing an arbitrary length output. This permutation was chosen exclusively because the published reference hardware implementation of the hash function implements the same rate and capacity.

2.2 Power Analysis Attacks

Power analysis attacks are a subclass of Side Channel Attack (SCA) in which the attacker has knowledge of the power consumption of the device under attack [21]. When cryptographic algorithms are implemented naïvely, variations in power consumption between operations can leak information. A subclass of these attacks, differential power analysis, can be used to attack algorithms which run repeatedly with a constant secret input or intermediate state by extracting the secret from thousands of runs for which those values remained constant [17].

2.3 Vulnerability of Keccak to Power Analysis Attacks

Power analysis attacks "do not exploit an inherent weakness of an algorithm," but rather characteristics of their implementation [10]. Prior work by Bertoni et. al. has shown that an unprotected implementation of the Keccak Sponge Function theoretically demonstrates distinguishability of power channel trace distributions with respect to input data [10]. Bertoni et. al. go on to show that a secret sharing algorithm can reduce and remove this distinguishability, and the algorithm they describe is the protection technique tested by this thesis.

2.4 Additivity and Secret Sharing

The hardening techniques to construct a power analysis-resistant implementation of the Keccak Sponge Function make use of a number of mathematical properties. First among these is the property of additivity (Definition 2.4.1), which is used to implement a technique called "secret sharing".

Definition 2.4.1. *Property of Additivity [13]*

Given a linear function H and two values A and B in the domain of H:

$$H(A + B) = H(A) + H(B)$$

The Keccak Sponge Function is linear for all operations except the permutation step χ [10]. If it is possible to create an implementation of Keccak which satisfies the properties of linearity, referred to hereafter as K', then Proposition 2.4.1 will hold.

Proposition 2.4.1. *Application of a Linear Sponge Function K' [10]*

Given a message M and a random bitstream N of length(M):

$$Keccak(M) = K'(M \oplus N) \oplus K'(N)$$

The $\oplus$ symbol represents bitwise exclusive or.

Definition 2.4.1 can be repeatedly applied to extend Proposition 2.4.1 to an arbitrary number of random bitstreams. In order to protect the algorithm against power analysis attacks, the computation of any one output share must be independent of at least one input share [10]. Bertoni et. al. showed that hardware implementations of Keccak require three shares (generating using two random bitstreams) to provide resistance to power analysis attacks [10]. The arguments to each instance of K' are then defined as "shares" as in Definition 2.4.2.

Definition 2.4.2. *Input Shares [10]*

Given a message M and two random bitstreams of length(M) N_1 and N_2:

$$A = M \oplus N_1 \oplus N_2$$

$$B = N_1$$

$$C = N_2$$

The sponge state for each share will be notated a, b, and c, corresponding to that share. The resulting message hash $Keccak(M)$ can be calculated by taking the bitwise exclusive or $K'(A) \oplus K'(B) \oplus K'(C)$. The power analysis attack-resistant sponge function created by this 3-share implementation is referred to as the Threshold Three-Share Implementation.

2.5 Threshold Three-Share Implementation of Keccak

Of the steps that make up the Keccak-f permutation function, χ is the only step which does not satisfy the property of additivity. To remedy this, Bertoni et. al. proposed a replacement for χ called χ' which is logically equivalent to χ when computed over the full set of sponge states {a, b, c} [10].

In the unprotected, single-share implementation of Keccak, χ is defined by Bertoni et. al. [10] as Definition 2.5.1.

Definition 2.5.1. *Single-Share Implementation of χ [10]*
Given sponge state a and row index $x \in [0 \dots 4]$:

$$a_{\chi\_out} = \chi(a) = a_x \oplus (a_{x+1} \oplus 1)a_{x+2}$$

This operation is modified by Bertoni et. al. [10] to result in χ', shown in Definition 2.5.2.

Definition 2.5.2. *Threshold Implementation of χ' [10]*
Given sponge states $\{a, b, c\}$ and row index $x \in [0 \dots 4]$:

$$a_{\chi\_out} = \chi'(b, c) = b_x \oplus (b_{x+1} \oplus 1)b_{x+2} \oplus b_{x+1}c_{x+2} \oplus b_{x+2}c_{x+1}$$
$$b_{\chi\_out} = \chi'(c, a) = c_x \oplus (c_{x+1} \oplus 1)c_{x+2} \oplus c_{x+1}a_{x+2} \oplus c_{x+2}a_{x+1}$$
$$c_{\chi\_out} = \chi'(a, b) = a_x \oplus (a_{x+1} \oplus 1)a_{x+2} \oplus a_{x+1}b_{x+2} \oplus a_{x+2}b_{x+1}$$

When χ' is substituted for χ in a three-share linearization of the Keccak sponge function, there is only one additional modification necessary to make the resulting computation result identical to a single-share non-linearized Keccak implementation. The ι step must be applied to only one of the three shares of the sponge function [10].

Definition 2.5.3. *Three-Share Implementation of ι [10]*

Given sponge states $\{a, b, c\}$ and without loss of generality:

$$a_{\iota\_out} = \iota(a)$$

$$b_{\iota\_out} = b$$

$$c_{\iota\_out} = c$$

2.6 Preserving Uniformity

Bilgin et. al. have shown that the threshold implementation of χ' is not sufficient for securing the Keccak Sponge Function against first-order differential power analysis [12]. They state that for a function f to be resistant to first-order DPA, it must be both non-complete, and uniform. Here, non-complete states that the output of f must be independent of at least one input share. This property is true of the threshold χ', as we see in Definition 2.5.2, where $a_{next} = \chi'(b, c)$. However, the threshold χ' does not preserve a uniform random distribution of input shares over the 24 rounds of the Keccak permutation function because it is not invertible (the step is not one-to-one, but takes multiple inputs to the same output) [12].

Bilgin et. al. preserve uniformity across the three shares through the injection of additional randomness during the χ step [12]. The scheme followed by this work is the one proposed in [12], where P and S are each 2-bit random vectors unique to each round of the permutation. χ' is as defined in Definition 2.5.2 and Bertoni et. al.'s prior work [10].

A uniformity-preserving implementation of χ' is shown in Definitions 2.6.1, 2.6.2, and 2.6.3 as defined by Bilgin et. al. [12].

Definition 2.6.1. *Uniform χ' for row $x \in [0 \dots 2]$ and column $y \in [0 \dots 4]$ [12]*

Given the threshold implementation of χ' as in Definition 2.5.2:

$$a_{\chi\text{-}out} = \chi'(b, c)$$

$$b_{\chi\text{-}out} = \chi'(c, a)$$

$$c_{\chi\text{-}out} = \chi'(a, b)$$

Definition 2.6.2. *Uniform χ' for row $x \in [3 \dots 4]$ and column $y = 0$ [12]*

Given the threshold implementation of χ' as in Definition 2.5.2:

$$a_{\chi\text{-}out} = \chi'(b, c) \oplus P_{x-2} \oplus S_{x-2}$$

$$b_{\chi\text{-}out} = \chi'(c, a) \oplus P_{x-2}$$

$$c_{\chi\text{-}out} = \chi'(a, b) \oplus S_{x-2}$$

Definition 2.6.3. *Uniform χ' for row $x \in [3 \dots 4]$ and column $y \in [1 \dots 4]$ [12]*

Given the threshold implementation of χ' as in Definition 2.5.2:

$$a_{\chi\text{-}out} = \chi'(b, c) \oplus a_{x,y-1} \oplus b_{x,y-1}$$

$$b_{\chi\text{-}out} = \chi'(c, a) \oplus a_{x,y-1}$$

$$c_{\chi\text{-}out} = \chi'(a, b) \oplus b_{x,y-1}$$

When this uniform implementation is substituted for χ in a three-share implementation of Keccak and ι is only applied to a single share, the result is again identical to the single-share, non-linearized implementation and the implementation is referred to as the Uniformity-Preserving Three-Share Implementation.

Chapter 3

IMPLEMENTATION IN-PRACTICE

The theory of protecting the Keccak Sponge Function against power analysis attack has been discussed and simulated by prior work [10, 12]. This work extends the analysis of these techniques to provide an independent verification of the correctness and resistance of the resulting algorithm. To accomplish this, a synthesizable VHDL implementation of the modified algorithm was created, starting with a public domain reference implementation of the algorithm.

3.1 Reference Implementation

With the theory of protecting the Keccak Sponge Function from power analysis attack established, the task of implementing and validating the behavior of the protection techniques can be discussed. This work bases the development of a protected hardware implementation on version 3.1 of the VHDL reference implementation published by Bertoni et. al. [11].

The published reference implementation implements the Keccak-f[r=1024,c=575] permutation, the sponge function used in the SHAKE128 XOF [14]. The reference offers the choice of a few different hardware accelerator designs. For this work, the "high speed core" design was chosen because it was suited to application of the protection techniques with minimal modifications to the structure of the implementation. The other design choices consume less chip area and power by serializing computation. For validating the protection techniques, optimizing for minimal chip area was not the highest priority. The FPGA test platform is not significantly constrained by design area or power consumption, and the modifications to the χ step are made

simpler by choosing the design with the least control signal overhead.

The reference implementation omits two steps of the full SHAKE128 XOF algorithm: input padding and output truncation. The input to the device must be delivered pre-padded using the pad10*1 scheme [14] in exactly rate-sized chunks, and the output of the function is delivered in exactly rate-sized chunks such that any output truncation must be performed by the consumer of the hardware implementation. For this work, the padding scheme was applied during input vector generation and a single 1024-bit output was taken as output, eliminating the need for the omitted steps in hardware.

The high speed core contains three main components: a finite state machine for driving the computation, a block which implements a single round of the permutation function, and a buffer of the sponge state.

3.2 Threshold Implementation

The Threshold Implementation of the Keccak Sponge Function is resistant to simple power analysis attacks (see Section 2.2 for a discussion of power analysis attack types). The first step of creating the threshold implementation is to implement the secret-sharing algorithm (Definition 2.4.2). A wrapper module, keccak_three_share, manages the random bit mixing in the secret sharing algorithm, the synchronization of control signals, and the configuration of the separate Keccak algorithm shares. The wrapper module provides the same interface as the original unprotected Keccak module, allowing for drop-in replacement of the unprotected algorithm.

Code. *Three-Share Secret Sharing in VHDL*

```
-- Input Share Computation
share_1 <= rand_1;
share_2 <= rand_2;
share_3 <= (rand_1 xor rand_2 xor din);

-- Output Share Recombination
dout <= (share_1_out xor share_2_out xor share_3_out);
```

For the purposes of test framework development, the "random" bitstream generation is performed by a 64-bit constant-seeded linear feedback shift register (LFSR). Such a generator is insufficient for resisting power analysis attack because the output is deterministic. However, this choice has a number of advantages for testing and validation. The simplicity of the LFSR allows for it to be instantiated multiple times with minimal effect on the consumption of FPGA area or power, resulting in efficiency and performance data which is minimally affected by the choice of random number generator. Also, though the bitstream is deterministic, the input vectors generated for the validation step of this work were created using random bytes from /dev/urandom on Unix. The result is that the deterministic bitstream generated by the LFSR is uncorrelated with the input bitstream, so our validation methodology followed is not affected by the choice of pseudo-random number generator (PRNG).

Within the secret sharing algorithm, three copies of the single share permutation function are instantiated and modified. For each share of the threshold implementation, χ' is a function of the output of the π step of the other two shares (Definition 2.5.2). Therefore, the instantiated single-share block was modified to output the result of the π step step from its permutation function and accept the input to the χ step of the other two shares. The single-share block was also modified with a boolean

iota-enable to allow the wrapper to selectively enable the ι step in only one instance of the algorithm as in Definition 2.5.3. The three instances were then interconnected as shown in Figure 3.1.

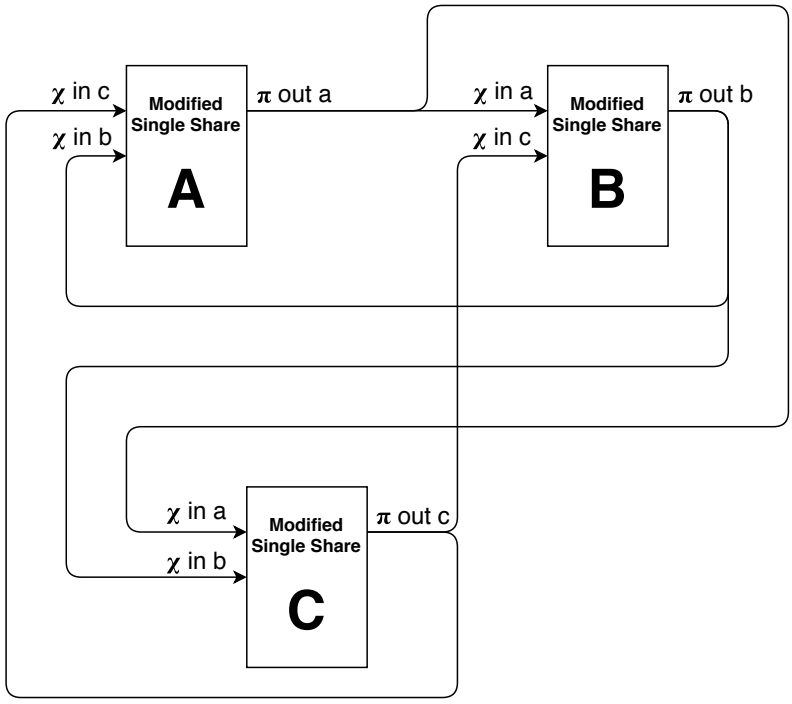

Figure 3.1: Interconnection of modified single-share blocks

The modified VHDL threshold implementation of the Keccak permutation round module can be seen in-full in the Appendix section A.1.

3.3 Achieving Uniformity

Definition 2.6.3 demonstrates that the uniform χ' is not symmetric with respect to the input shares because bits from shares A and B are re-injected to preserve uniformity. The uniform χ step is placed as its own submodule in the three-share wrapper, resulting is an interconnection scheme as shown in Figure 3.2.

The VHDL implementation of the three-share uniform χ' can be seen in-full in

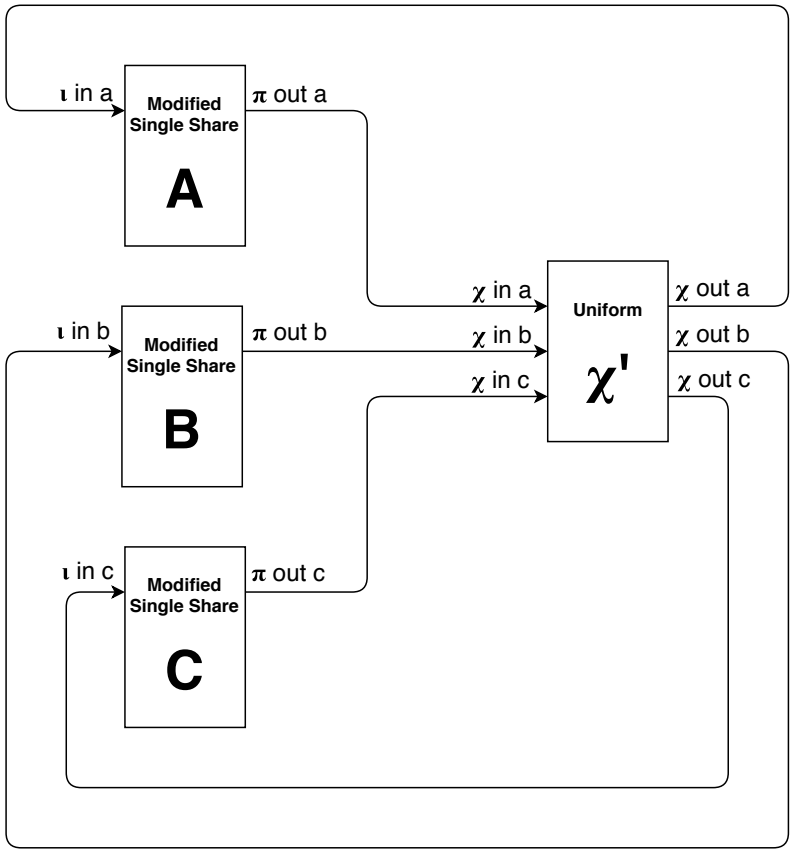

Figure 3.2: Interconnection of uniformity-preserving implementation

the Appendix section A.2.

3.4 Test Framework

The reference implementation of the Keccak Sponge Function is a hardware accelerator and must be driven by a top-level block which handles the accelerator's control signals, inputs, and outputs. These tasks were accomplished by the development of a top-level block which incorporated the following elements:

- A clock frequency resampler to adjust the clock frequency, allowing the design to meet power trace measurement bandwidth requirements

- A block memory for storing input test vectors
- A USB UART so that the device could be controlled by a PC over USB.

The resulting platform was synthesized using Xilinx Vivado WebPACK 2015.2 [5] for a Digilent Nexys 4 DDR FPGA Development Board [3], featuring a Xilinx Artix-7 FPGA [1].

Chapter 4

EFFICIENCY AND PERFORMANCE

4.1 Synthesis Results for FPGA

The test platform was synthesized for a Digilent Nexys 4 DDR FPGA Development Board, featuring a Xilinx Artix-7 FPGA using Xilinx Vivado 2015.2. To estimate the FPGA resource consumption of the designs, measurements of the area and power characteristics were collected from the Vivado design report after bitstream generation. The unprotected single-share, threshold three-share, and uniform three-share implementations were synthesized using various optimization settings. The optimization strategies for each design are referred to in shorthand as described in Table 4.1.

Utilization of the Artix-7 FPGA is measured in flip-flop and look-up table slices as reported by Vivado after the Implementation step. Figure 4.1 displays the utilization of each implementation, normalized to the total number of slices used by the single-share defaults-optimized implementation.

Power consumption is as reported by Vivado after the Implementation step. Similarly as in Figure 4.1, the reported values in Figure 4.2 are normalized to the on-chip power consumption of the single-share defaults-optimized implementation.

Table 4.1: Optimization Strategies

Optimization Strategy Shorthand	Synthesis Setting	Implementation Setting
Defaults	Defaults	Defaults
Optimize Area	Flow_AreaOptimized_High	Area_Explore
Optimize Performance	Flow_PerfOptimized_High	Performance_Explore
Optimize Power	Flow_AreaOptimized_High	Power_DefaultOpt

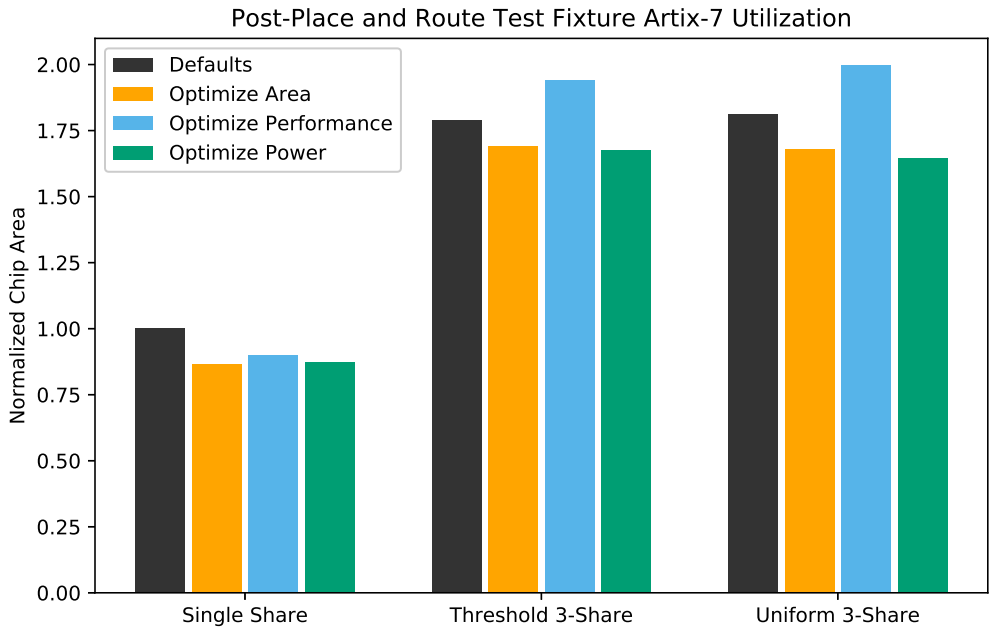

Figure 4.1: Comparing chip area utilization for different implementations and optimization strategies of the Keccak Sponge Function

4.2 Analysis of Synthesis Results

Though the techniques described to implement the three-share logic represent three parallel implementations of the Keccak sponge function, the design only consumes approximately double the chip area of the baseline implementation. However, the three-share design does consume up to three times the power of the single-share algorithm.

Important to note is that the data presented here does not fully represent the additional resource consumption required for the implementation of a protected implementation of Keccak. The additional demand of high-entropy random bitstream generation will increase the demand on area and power. For this work, the random number generation is approximated by a LFSR as discussed in Section 3.2. For the

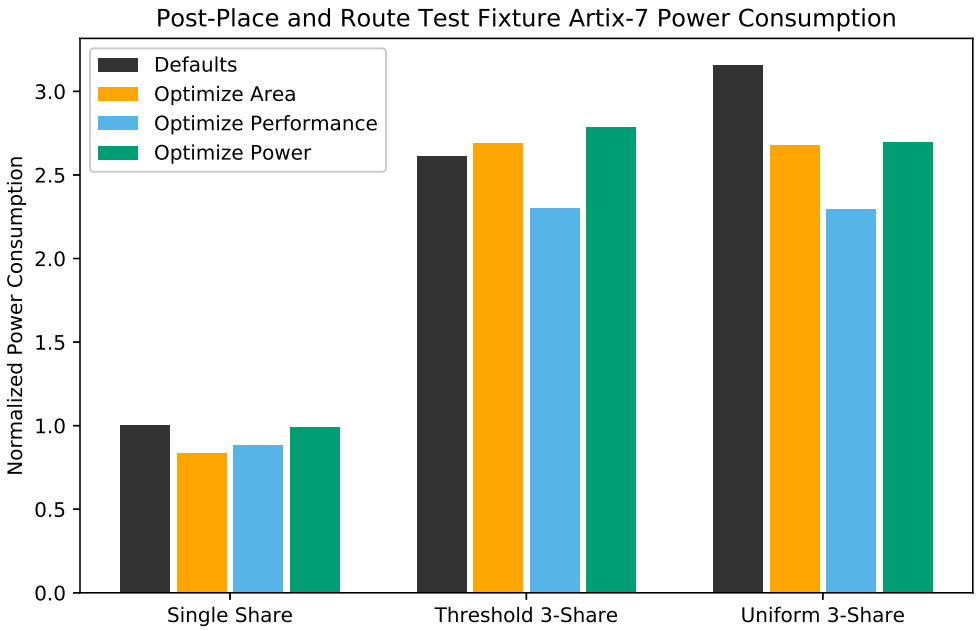

Figure 4.2: Comparing on-chip power consumption for different implementations and optimization strategies of the Keccak Sponge Function

protection mechanisms to withstand an actuall attack, the LFSR must be replaced by a CSPRNG, because if the random bitstream can be predicted by an attacker then then input masking is no longer effective [10].

Because the power consumption roughly triples while the area roughly doubles, it's more likely that the inclusion of a protected hardware implementation will be constrained by the available power supply or by chip heat dissipation before chip area is exhausted. Regardless, the architecture presented in this paper is structured specifically for throughput, not for efficiency, and the data presented here shows that the cost of protecting a high-speed hardware accelerator for the Keccak Sponge Function against power analysis attack is considerable.

Future work on implementation of the protected Keccak Sponge Function has room to expand in this domain. Work by Bertoni et. al. does describe a serial ar-

chitecture instead of the parallel design implemented in this work [10]. This would potentially allow for a single instance of the modified single-share algorithm to compute the result without leaking information through the power channel at the cost of taking three times as many cycles to complete the computation.

Chapter 5

VALIDATION OF POWER ANALYSIS ATTACK RESISTANCE

5.1 Testing Methodology

To validate the techniques to harden the Keccak sponge function against power analysis attacks, I used the testing methodology described by Goodwill et. al [15]. In this section, I'll provide an overview of how I adapted the methodology to test my implementation of the Keccak Sponge Function.

The original methodology seeks to provide a pass/fail criterion for determining whether a device under test (DUT) exhibits a maximum allowed correlation between intermediate state during computation and fluctuations in the DUT's power consumption. This threshold, if exceeded, indicates that the DUT is vulnerable to power analysis attack. There are three steps to the validation methodology:

1. DUT power trace measurement
2. DUT input selection and partitioning
3. T-statistic calculation and analysis

5.1.1 DUT Set-Up and Power Trace Measurement

The unprotected, protected three-share, and protected three-share uniformity-preserving implementations of the Keccak Sponge Function were synthesized for the Xilinx Artix-7 FPGA on the Digilent Nexys 4 DDR FPGA Development Board as discussed in Chapter 4. This platform was chosen because of the availability and low cost of prototyping digital logic on FPGA platforms and because the development board supplied

a USB UART for communication with a host computer for driving the validation test data collection.

In the original methodology, the power trace is measured differentially across a shunt resistor in series with the device under test. The Digilent Nexys 4 DDR Development Board was powered over the USB connection with the host computer, and the FPGA, behaving as the DUT, was powered by a collection of LDO DC-DC power regulators on the development board.

The design of the development board did not readily facilitate the addition of a series shunt resistor to allow for the methodology-suggested power trace measurement. However, the development board did provide a broken-out test point for the FPGA's primary $3.3V$ supply voltage (VCCO) at pad J11. I measured the voltage relative to ground at test point J11. The collected power traces from this test measure the voltage drop across the DUT in series with the LDO power regulator.

To maximize the voltage fluctuation relative to the power consumption of the FPGA at test point J11, I used a hot air rework station to remove all filter caps with nominal value equal to or greater than $1\mu F$ connected across VCCO and ground. On the Digilent Nexys 4 DDR, this included the following capacitors [4]:

- C86, C87, C88
- C98, C99, C100
- C122, C123, C124
- C127
- C147
- C180

The capacitors are located on the underside of the Digilent Nexys 4 DDR. A picture of the board after capacitor removal is included in Figure 5.1.

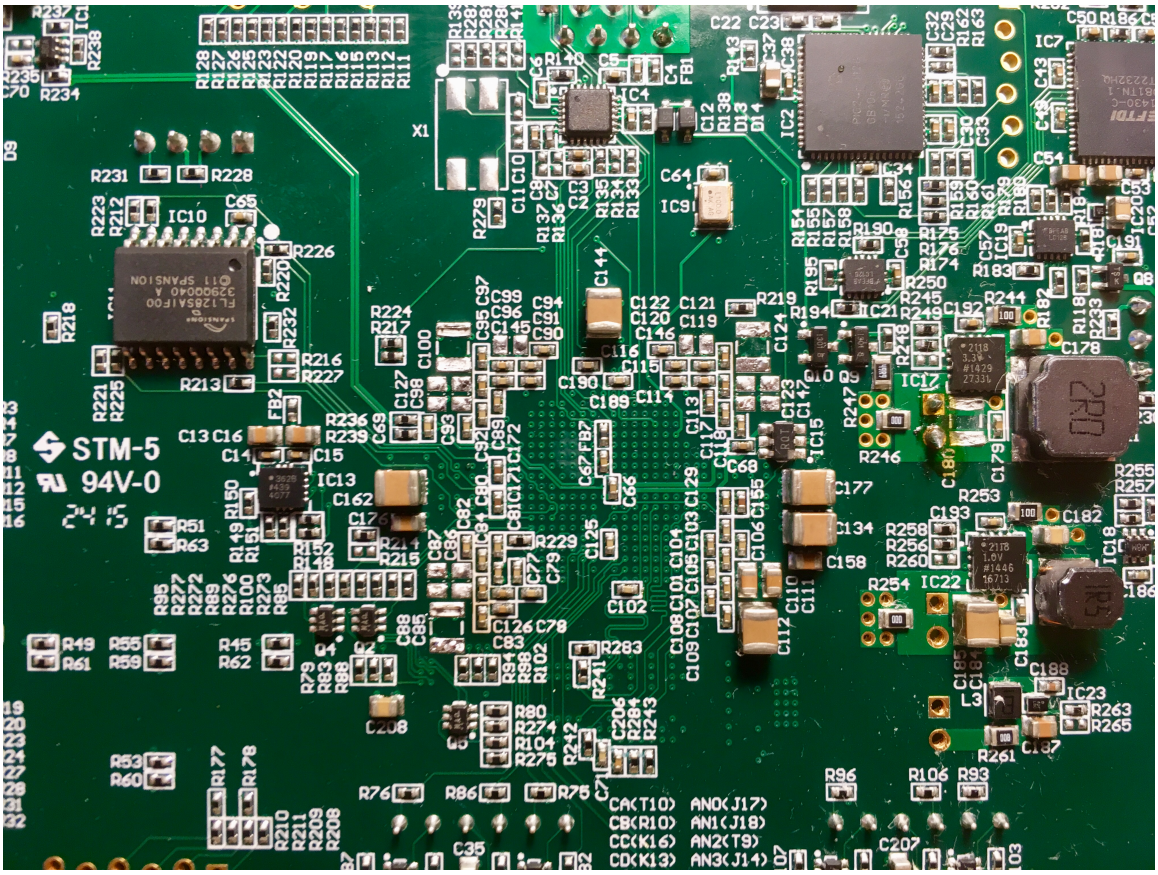

Figure 5.1: Underside of the Nexys 4 DDR after capacitor removal

The original methodology specifies that the power traces be measured by an oscilloscope or other A/D measurement apparatus with the following properties [15]:

1. Bandwidth of at least 50% of the device clock rate for software implementations and at least 80% of the clock rate for hardware implementations
2. Capability to capture samples at 5x the bandwidth
3. A minimum of 8bits of sampling resolution
4. Enough storage to capture the entire signal required for the test and analysis

To meet these requirements I measured the power traces using a Keysight InfiniiVision MSO-X 2022A Mixed Signal Oscilloscope with a 200 MHz maximum bandwidth and 2 GSa/s maximum sample rate [2]. Channel 1 was connected to test point J11 and the external trigger was connected to PMOD JA Pin 1 on the development board. The native clock speed of the Digilent Nexys 4 Development Board is 100 MHz. To increase the fidelity of the measured power traces relative to the sponge function clock rate, the Xilinx Artix-7 FPGA was internally clocked down to 10 MHz using the Xilinx Clock Wizard IP [23].

A finite state machine encapsulated the Keccak Sponge Function on the FPGA and coordinated communication with the host computer over the USB UART interface. The DUT emits a pulse on a GPIO output to provide a trigger signal to the oscilloscope, resulting in synchronized power traces. The following algorithm represents the power trace collection procedure in pseudocode. The complete Python script for driving the trace collection can be found in Appendix A.3.

```
Result: A set of power traces
openFpga();
openScope();
openDatabase();
configureOscilloscope();
numTraces = 0;
while numTraces < desiredNumberOfTraces do
    if !fixedInput then
        input = getRandomBytes(126);
    else
        input = SELECTED_FIXED_INPUT;
    end
    paddedInput = pad101(input);
    expectedOutput = SHAKE128(input);
    startScopeCapture();
    fpgaOutput = sendReceive(paddedInput);
    trace = getScopeTrace();
    if expectedOutput == fpgaOutput then
        writeToDatabase(input, trace);
        numTraces += 1;
    end
end
```

Algorithm 1: Power Trace Collection Procedure

To facilitate synchronization and alignment of power traces, the finite state machine outputs a trigger edge at the beginning and end of the sponge function permutation on PMOD JA Pin 1.

5.1.2 DUT Input Selection and Partitioning

In the testing methodology, the input to the DUT is partitioned into two data sets. Data set A consists of the power traces collected for at least 5000 randomly generated inputs, and data set B consists of the power traces for at least 5000 runs of a single, fixed input. For the fixed-input I selected, at random, the value (in hex):

```
5c2c 43fe c6a3 87d8 763b 79af 7ca2 d038
441b ac29 5074 9df2 3a4c 1ee6 7ccb a9a7
0019 5a70 864a 557f cc82 9bde 0762 3218
946f 243f b96f 9478 d840 689a 8462 12e5
1296 76ac 64c0 91de b523 1c17 ec92 b4ef
84f1 a242 e26f 50ce 11e6 5b34 ced4 5034
3fdf 2ee6 97d6 f2f1 c05e 4e16 b816 a21c
97eb 152c 4625 aed2 62ed 59b6 ee58
```

An arbitrary fixed input is appropriate for this test, because the ability to distinguish different inputs to the function based on the power trace of the DUT represents a vulnerability of the DUT to power analysis attacks. For each data partition, for each implementation, I collected at least $100,000$ traces.

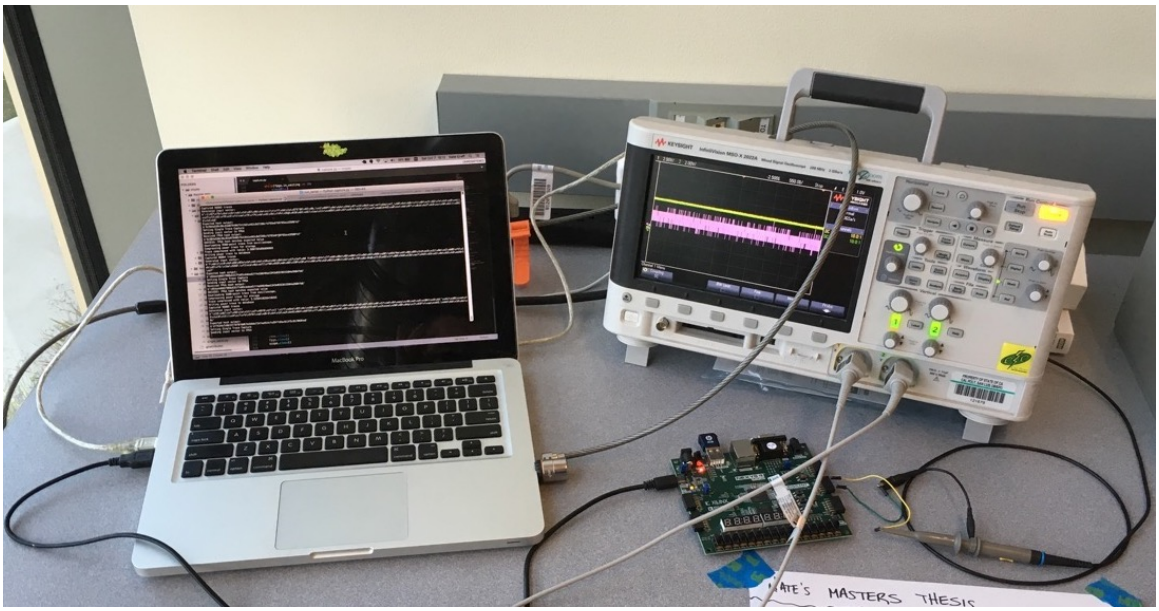

Figure 5.2: Experimental set-up for power trace collection

5.1.3 T-Statistic Calculation and Analysis

According to the testing methodology, the resulting power traces are combined into a trace called the "T-statistic" using the following algorithm point-wise [15]:

Data:

Symbol	Description
X_A	The point-average of all traces in data set A
X_B	The point-average of all traces in data set B
S_A	The point-standard deviation of all traces in data set A
S_B	The point-standard deviation of all traces in data set B
N_A	The number of traces in data set A
N_B	The number of traces in data set B

Result: T, The T-Statistic Trace for a DUT

$$T = \frac{X_A - X_B}{\sqrt{\frac{S_A^2}{N_A} + \frac{S_B^2}{N_B}}}$$

Algorithm 2: T-Statistic Calculation

The pass/fail criteria for a DUT is whether the T-statistic trace exceeds a threshold of $\pm C$ at any point along the trace. The higher the value of C, the more correlation is permitted between the power trace and the input data for the device to pass the validation criteria.

For the purposes of validating the power analysis resistance of the protected implementations of the Keccak Sponge Function, I compared the maximum absolute values for the T-statistics corresponding to each of the Keccak implementations. To show that the protected implementations significantly limit the correlation between input data and power trace, I show that the maximum excursion of the T-statistic for each of the protected implementations is much lower than the maximum excursion of the T-statistic for the unprotected implementation.

5.2 Validation Results

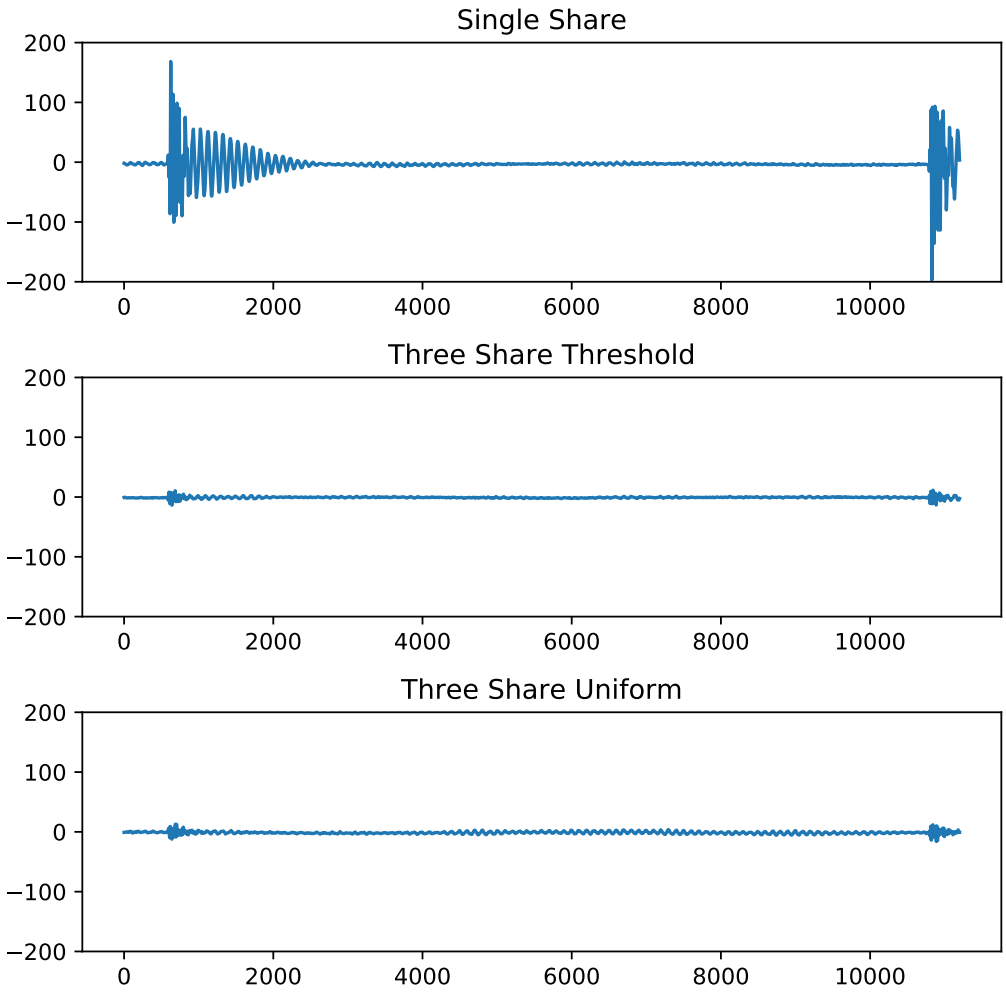

Figure 5.3: T-statistics over 5000 collected power traces

Table 5.1: T-Statistic Maximum Excursions

Implementation	Unprotected	Three-Share	Uniformity-Preserving
Maximum $\|T\|$	197.209	13.766	15.939
Normalized to Unprotected	1	0.0698	0.0808

The T-statistics for each of the implementations of the Keccak Sponge Function are shown in Figure 5.3 and their respective maximum excursions in Table 5.1. The result clearly shows that the protected implementations meet a much more strict criteria for power analysis validation. The unprotected implementation will fail validation under these conditions for any maximum threshold $|C| <= 197.209$, whereas the protected implementations reduce the maximum $|C|$ validation criterion to 15.939. This represents more than a 12-fold decrease in maximum tolerable $|C|$ for power analysis validation of the sponge function, showing that the protected implementations do significantly improve the resistance of the sponge function to power analysis-based side-channel attack.

Chapter 6

CONCLUSION

This work demonstrates that post-synthesis implementation of the power channel protection techniques described by Bertoni. et. al and Bilgin et. al. for the Keccak Sponge Function measurably decrease the correlation between the data input to the function and the power consumption of the algorithm. This extends the results of prior work to show that the techniques do not only work in simulation, but in practice on a Xilinx Artix-7 FPGA. Additionally, this represents an open-source publication of a functional HDL implementation of the Keccak Sponge Function which is resistant to simple and differential power analysis attack, combined with measurements demonstrating the effect of the protection techniques on design resources including chip area and power consumption as well as the Rambus validation methodology test results which show that the published implementation is protected against power analysis attack.

BIBLIOGRAPHY

[1] Artix-7 fpga family. https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html.

[2] Keysight infiniivision 2000 x-series. https://literature.cdn.keysight.com/litweb/pdf/5990-6679EN.pdf?id=1999123.

[3] Nexys 4 ddr. https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start.

[4] Nexys 4 ddr schematic. https://reference.digilentinc.com/learn/documentation/schematics/nexys-4-ddr-schematic.

[5] Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html.

[6] E. Andreeva, B. Mennink, B. Preneel, and M. Škrobot. Security analysis and comparison of the sha-3 finalists blake, grøstl, jh, keccak, and skein. In *International Conference on Cryptology in Africa*, pages 287–305. Springer, 2012.

[7] J.-P. Aumasson and W. Meier. Zero-sum distinguishers for reduced keccak-f and for the core functions of luffa and hamsi. *rump session of Cryptographic Hardware and Embedded Systems-CHES*, 2009:67, 2009.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007. Citeseer, 2007.

[9] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3(30), 2009.

[10] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Building power analysis resistant implementations of keccak. In *Second SHA-3 candidate conference*, volume 142. Citeseer, 2010.

[11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Hardware implementation in vhdl. `https://keccak.team/archives.html`, 2016.

[12] B. Bilgin, J. Daemen, V. Nikov, S. Nikova, V. Rijmen, and G. Van Assche. Efficient and first-order dpa resistant implementations of keccak. In *International Conference on Smart Card Research and Advanced Applications*, pages 187–199. Springer, 2013.

[13] M. J. Bradley and D. L. Finn. Additivity + homogeneity. *College Mathematics Journal*, 30(2):133–135, March 1999.

[14] P. FIPS. Secure hash algorithm-3 (sha-3) standard: Permutation-based hash and extendable-output functions. *National Institute for Standards and Technology (NIST)*, 202(0), 2014.

[15] B. J. Gilbert Goodwill, J. Jaffe, P. Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, 2011.

[16] J. Guo, M. Liu, and L. Song. Linear structures: Applications to cryptanalysis of round-reduced keccak. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 249–274. Springer, 2016.

[17] R. McEvoy, M. Tunstall, C. C. Murphy, and W. P. Marnane. Differential power analysis of hmac based on sha-2, and countermeasures. In *International Workshop on Information Security Applications*, pages 317–332. Springer, 2007.

[18] P. Morawiecki, J. Pieprzyk, and M. Srebrny. Rotational cryptanalysis of round-reduced keccak. In *International Workshop on Fast Software Encryption*, pages 241–262. Springer, 2013.

[19] P. Morawiecki and M. Srebrny. A sat-based preimage analysis of reduced keccak hash functions. *Information Processing Letters*, 113(10-11):392–397, 2013.

[20] B. Schneier. One-way hash functions. *Applied Cryptography, Second Edition, 20th Anniversary Edition*, pages 429–459, 1996.

[21] B. Schneier. Cryptographic design vulnerabilities. *Computer*, 31(9):29–33, 1998.

[22] Xilinx. Block memory generator. `https://www.xilinx.com/products/intellectual-property/block_memory_generator.html`, February 2017.

[23] Xilinx. Clocking wizard. `https://www.xilinx.com/products/intellectual-property/clocking_wizard.html`, February 2017.

APPENDICES

Appendix A

CODE LISTINGS

A.1 Threshold Round Permutation Module

```
-- The Keccak sponge function, designed by Guido Bertoni, Joan Daemen,
-- Michal Peeters and Gilles Van Assche. For more information, feedback or
-- questions, please refer to our website: http://keccak.noekeon.org/

-- Implementation by the designers,
-- hereby denoted as "the implementer".

-- To the extent possible under law, the implementer has waived all copyright
-- and related or neighboring rights to the source code in this file.
-- http://creativecommons.org/publicdomain/zero/1.0/

library work;
        use work.keccak_globals.all;

library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_arith.all;


entity keccak_round_multi_share is
port (
    round_in              : in  k_state;
    round_constant_signal : in  std_logic_vector(63 downto 0);
    pi_state_out          : out k_state;
    iota_state_in         : in  k_state;
    iota_en               : in  std_logic;
    round_out             : out k_state);
end keccak_round_multi_share;

architecture rtl of keccak_round_multi_share is

-------------------------------------------------------------------------------
-- Internal signal declarations
-------------------------------------------------------------------------------

signal theta_in, theta_out, pi_in, pi_out, rho_in, rho_out, iota_in, iota_out : k_state;
signal sum_sheet: k_plane;

begin  -- Rtl

    --connecitons
```

```
--order theta, pi, rho, chi, iota
theta_in      <= round_in;
rho_in        <= theta_out;
pi_in         <= rho_out;
-- route chi out of the share for uniformity
pi_state_out  <= pi_out; -- input to chi
iota_in       <= iota_state_in; -- output from chi
round_out     <= iota_out;

--theta

--compute sum of columns

i0101: for x in 0 to 4 generate
    i0102: for i in 0 to 63 generate
        sum_sheet(x)(i)<=theta_in(0)(x)(i) xor theta_in(1)(x)(i) xor theta_in(2)(x)(i) xor theta_in(3)(x)(i) xo

    end generate;
end generate;


i0200: for y in 0 to 4 generate
    i0201: for x in 1 to 3 generate
        theta_out(y)(x)(0)<=theta_in(y)(x)(0) xor sum_sheet(x-1)(0) xor sum_sheet(x+1)(63);
        i0202: for i in 1 to 63 generate
            theta_out(y)(x)(i)<=theta_in(y)(x)(i) xor sum_sheet(x-1)(i) xor sum_sheet(x+1)(i-1);
        end generate;
    end generate;
end generate;

i2001: for y in 0 to 4 generate
    theta_out(y)(0)(0)<=theta_in(y)(0)(0) xor sum_sheet(4)(0) xor sum_sheet(1)(63);
    i2021: for i in 1 to 63 generate
        theta_out(y)(0)(i)<=theta_in(y)(0)(i) xor sum_sheet(4)(i) xor sum_sheet(1)(i-1);
    end generate;

end generate;

i2002: for y in 0 to 4 generate
    theta_out(y)(4)(0)<=theta_in(y)(4)(0) xor sum_sheet(3)(0) xor sum_sheet(0)(63);
    i2022: for i in 1 to 63 generate
        theta_out(y)(4)(i)<=theta_in(y)(4)(i) xor sum_sheet(3)(i) xor sum_sheet(0)(i-1);
    end generate;

end generate;

-- pi
i3001: for y in 0 to 4 generate
    i3002: for x in 0 to 4 generate
        i3003: for i in 0 to 63 generate
            --pi_out(y)(x)(i)<=pi_in((y +2*x) mod 5)(((4*y)+x) mod 5)(i);
            pi_out((2*x+3*y) mod 5)(0*x+1*y)(i)<=pi_in(y) (x)(i);
        end generate;
    end generate;
```

```
end generate;

--rho


i4001: for i in 0 to 63 generate
    rho_out(0)(0)(i)<=rho_in(0)(0)(i);
end generate;
i4002: for i in 0 to 63 generate
    rho_out(0)(1)(i)<=rho_in(0)(1)((i-1)mod 64);
end generate;
i4003: for i in 0 to 63 generate
    rho_out(0)(2)(i)<=rho_in(0)(2)((i-62)mod 64);
end generate;
i4004: for i in 0 to 63 generate
    rho_out(0)(3)(i)<=rho_in(0)(3)((i-28)mod 64);
end generate;
i4005: for i in 0 to 63 generate
    rho_out(0)(4)(i)<=rho_in(0)(4)((i-27)mod 64);
end generate;

i4011: for i in 0 to 63 generate
    rho_out(1)(0)(i)<=rho_in(1)(0)((i-36)mod 64);
end generate;
i4012: for i in 0 to 63 generate
    rho_out(1)(1)(i)<=rho_in(1)(1)((i-44)mod 64);
end generate;
i4013: for i in 0 to 63 generate
    rho_out(1)(2)(i)<=rho_in(1)(2)((i-6)mod 64);
end generate;
i4014: for i in 0 to 63 generate
    rho_out(1)(3)(i)<=rho_in(1)(3)((i-55)mod 64);
end generate;
i4015: for i in 0 to 63 generate
    rho_out(1)(4)(i)<=rho_in(1)(4)((i-20)mod 64);
end generate;

i4021: for i in 0 to 63 generate
    rho_out(2)(0)(i)<=rho_in(2)(0)((i-3)mod 64);
end generate;
i4022: for i in 0 to 63 generate
    rho_out(2)(1)(i)<=rho_in(2)(1)((i-10)mod 64);
end generate;
i4023: for i in 0 to 63 generate
    rho_out(2)(2)(i)<=rho_in(2)(2)((i-43)mod 64);
end generate;
i4024: for i in 0 to 63 generate
    rho_out(2)(3)(i)<=rho_in(2)(3)((i-25)mod 64);
end generate;
i4025: for i in 0 to 63 generate
    rho_out(2)(4)(i)<=rho_in(2)(4)((i-39)mod 64);
end generate;

i4031: for i in 0 to 63 generate
    rho_out(3)(0)(i)<=rho_in(3)(0)((i-41)mod 64);
```

```
    end generate;
    i4032: for i in 0 to 63 generate
        rho_out(3)(1)(i)<=rho_in(3)(1)((i-45)mod 64);
    end generate;
    i4033: for i in 0 to 63 generate
        rho_out(3)(2)(i)<=rho_in(3)(2)((i-15)mod 64);
    end generate;
    i4034: for i in 0 to 63 generate
        rho_out(3)(3)(i)<=rho_in(3)(3)((i-21)mod 64);
    end generate;
    i4035: for i in 0 to 63 generate
        rho_out(3)(4)(i)<=rho_in(3)(4)((i-8)mod 64);
    end generate;

    i4041: for i in 0 to 63 generate
        rho_out(4)(0)(i)<=rho_in(4)(0)((i-18)mod 64);
    end generate;
    i4042: for i in 0 to 63 generate
        rho_out(4)(1)(i)<=rho_in(4)(1)((i-2)mod 64);
    end generate;
    i4043: for i in 0 to 63 generate
        rho_out(4)(2)(i)<=rho_in(4)(2)((i-61)mod 64);
    end generate;
    i4044: for i in 0 to 63 generate
        rho_out(4)(3)(i)<=rho_in(4)(3)((i-56)mod 64);
    end generate;
    i4045: for i in 0 to 63 generate
        rho_out(4)(4)(i)<=rho_in(4)(4)((i-14)mod 64);
    end generate;

    --iota
    i5001: for y in 1 to 4 generate
        i5002: for x in 0 to 4 generate
            i5003: for i in 0 to 63 generate
                iota_out(y)(x)(i)<=iota_in(y)(x)(i);
            end generate;
        end generate;
    end generate;

        i5012: for x in 1 to 4 generate
            i5013: for i in 0 to 63 generate
                iota_out(0)(x)(i)<=iota_in(0)(x)(i);
            end generate;
        end generate;

            i5103: for i in 0 to 63 generate
                iota_out(0)(0)(i)<=iota_in(0)(0)(i) xor (round_constant_signal(i) and iota_en);
            end generate;

end rtl;
```

A.2 Uniform Chi

```
----------------------------------------------------------------------------------
-- Group: Cal Poly CPE SHA-3 Research Team
-- Engineer: Nathaniel Graff
--
-- Create Date: 12/06/2016 03:38:12 PM
-- Design Name: chi_uniform
-- Module Name: chi_uniform - Behavioral
-- Project Name: Keccak Research
-- Description: Uniform three-share chi step implementation
--
----------------------------------------------------------------------------------

library work;
  use work.keccak_globals.all;

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;

entity chi_uniform is
port(
  chi_rand_bits : in  std_logic_vector(3 downto 0);
  chi_in_a      : in  k_state;
  chi_in_b      : in  k_state;
  chi_in_c      : in  k_state;
  chi_out_a     : out k_state;
  chi_out_b     : out k_state;
  chi_out_c     : out k_state);
end chi_uniform;

architecture Behavioral of chi_uniform is

  -- Intermediate values to store the output of chi prime
  signal chi_prime_a, chi_prime_b, chi_prime_c       : k_state;

  -- Random bits
  signal p, s : std_logic_vector(1 downto 0);

begin

  -- Map random bits into p and s for easy consumption
  p <= chi_rand_bits(1 downto 0);
  s <= chi_rand_bits(3 downto 2);

  -- Chi prime
  i0000: for y in 0 to 4 generate
    i0001: for x in 0 to 4 generate
      i0002: for i in 0 to 63 generate
        chi_prime_a(y)(x)(i) <= chi_in_b(y)(x)(i) xor
                    ((not chi_in_b(y)((x+1) mod 5)(i)) and chi_in_b(y)((x+2) mod 5)(i)) xor
                    (chi_in_b(y)((x+1) mod 5)(i) and chi_in_c(y)((x+2) mod 5)(i)) xor
                    (chi_in_b(y)((x+2) mod 5)(i) and chi_in_c(y)((x+1) mod 5)(i));
```

```
        chi_prime_b(y)(x)(i) <= chi_in_c(y)(x)(i) xor
                    ((not chi_in_c(y)((x+1) mod 5)(i)) and chi_in_c(y)((x+2) mod 5)(i)) xor
                    (chi_in_c(y)((x+1) mod 5)(i) and chi_in_a(y)((x+2) mod 5)(i)) xor
                    (chi_in_c(y)((x+2) mod 5)(i) and chi_in_a(y)((x+1) mod 5)(i));

        chi_prime_c(y)(x)(i) <= chi_in_a(y)(x)(i) xor
                    ((not chi_in_a(y)((x+1) mod 5)(i)) and chi_in_a(y)((x+2) mod 5)(i)) xor
                    (chi_in_a(y)((x+1) mod 5)(i) and chi_in_b(y)((x+2) mod 5)(i)) xor
                    (chi_in_a(y)((x+2) mod 5)(i) and chi_in_b(y)((x+1) mod 5)(i));
      end generate;
    end generate;
  end generate;

  -- Propogating rows not included in further steps
  i0010: for y in 0 to 4 generate
    i0011: for x in 0 to 2 generate
      i0012: for i in 0 to 63 generate
        chi_out_a(y)(x)(i) <= chi_prime_a(y)(x)(i);

        chi_out_b(y)(x)(i) <= chi_prime_b(y)(x)(i);

        chi_out_c(y)(x)(i) <= chi_prime_c(y)(x)(i);
      end generate;
    end generate;
  end generate;

  -- Injection of random bits
    i0021: for x in 3 to 4 generate
      i0022: for i in 0 to 63 generate
        chi_out_a(0)(x)(i) <= chi_prime_a(0)(x)(i) xor p(x-3) xor s(x-3);

        chi_out_b(0)(x)(i) <= chi_prime_b(0)(x)(i) xor p(x-3);

        chi_out_c(0)(x)(i) <= chi_prime_c(0)(x)(i) xor s(x-3);
      end generate;
    end generate;

  -- Mixing to jointly satisfy uniformity
  i0030: for y in 1 to 4 generate
    i0031: for x in 3 to 4 generate
      i0032: for i in 0 to 63 generate
        chi_out_a(y)(x)(i) <= chi_prime_a(y)(x)(i) xor chi_in_a(y-1)(x)(i) xor chi_in_b(y-1)(x)(i);

        chi_out_b(y)(x)(i) <= chi_prime_b(y)(x)(i) xor chi_in_a(y-1)(x)(i);

        chi_out_c(y)(x)(i) <= chi_prime_c(y)(x)(i) xor chi_in_b(y-1)(x)(i);
      end generate;
    end generate;
  end generate;

end Behavioral;
```

A.3 Power Trace Capture Script

```
#!/usr/bin/env python3

import serial
import visa
import psycopg2
import sys
import os
import codecs
import lzma
import matplotlib.pyplot as pyplot
from time import sleep
from tabulate import tabulate
import keccak

FPGA_PORT = ""

DBHOST = ""
DBNAME = ""
DBUSER = ""
DBPASS = ""

CAPTURE_TRACE_COUNT = 100000
CONFIG_NUMBER = 0
ENABLE_STATIC = False
STATIC_INPUT = b'\\,C\xfe\xc6\xa3\x87\xd8v;y\xaf|\xa2\xd08D\x1b\xac)Pt\x9d' + \
               b'\xf2:L\x1e\xe6|\xcb\xa9\xa7\x00\x19Zp\x86JU\x7f\xcc\x82' + \
               b'\x9b\xde\x07b2\x18\x94o$?\xb9o\x94x\xd8@h\x9a\x84b\x12\xe5' + \
               b'\x12\x96v\xacd\xc0\x91\xde\xb5#\x1c\x17\xec\x92\xb4\xef' + \
               b'\x84\xf1\xa2B\xe2oP\xce\x11\xe6[4\xce\xd4P4?\xdf.\xe6\x97' + \
               b'\xd6\xf2\xf1\xc0^N\x16\xb8\x16\xa2\x1c\x97\xeb\x15,F%\xae' + \
               b'\xd2b\xedY\xb6\xeeX'

def getScope():
    rm = visa.ResourceManager()
    instruments = rm.list_resources()

    if(len(instruments) == 0):
        print("No instruments connected")
        exit()
    elif(len(instruments) == 1):
        scope = rm.open_resource(instruments[0])
        print("Connected to instrument: " + instruments[0])
    else:
        print("Please select an instrument:")
        for (num, instrument) in zip(range(0, len(instruments)), instruments):
            print(str(num) + " -- " + instruments[num])
        sys.stdout.write("Instrument number: ")
        num = int(sys.stdin.read(1))
        scope = rm.open_resource(instruments[num])

    scope.timeout = 10000
    scope.read_termination = '\n'
```

```
    scope.write_termination = '\n'
    scope.clear()

    print("Querying scope identity...")
    instrumentdata = scope.query("*IDN?").split(',')
    print(tabulate([instrumentdata], ["Manufacturer", "Model", "Serial #", "Software Version"], tablefmt="psql"))

    scope.write("*RST")
    scope.query("*OPC?")

    return scope

def scopeCommand(scope, command):
    scope.write(command)
    scope.query("*OPC?")

def getFPGA():
    return serial.Serial(FPGA_PORT, timeout=1)

def rearrangeHex(bin):
    # reverse byte order within each word
    bout = b''
    for word in [bin[16*n:16*(n+1)] for n in range(len(bin) // 16)]:
        bytelist = [word[2*n:2*(n+1)] for n in range(len(word) // 2)]
        bytelist.reverse()
        bout += b''.join(bytelist)
    return bout

def rearrangeBytes(bin):
    return codecs.decode(rearrangeHex(codecs.encode(bin, 'hex_codec')), 'hex_codec')

def getTrace(scope):
    scope.query("*OPC?")

    trace = b''
    scopeCommand(scope, ":WAVeform:SOURce CHANnel1")
    scopeCommand(scope, ":WAVeform:FORMat BYTE")
    scopeCommand(scope, ":WAVeform:UNSigned ON")
    scopeCommand(scope, ":WAVeform:BYTeorder MSBFirst")
    scopeCommand(scope, ":WAVeform:POINts MAXimum")

    length = int(scope.query(":WAVeform:POINts?"))
    print("Downloading " + str(length) + "-point trace from oscilloscope.")

    scope.write(":WAVeform:DATA?")
    trace += scope.read_raw()

    return trace[10:-1]

def configCapture(scope):
    scope.query("*OPC?")

    # turn on both channels
    scopeCommand(scope, ":CHANnel1:DISPlay ON")
```

```
    # trigger on the positive edge of the external trigger input
    scopeCommand(scope, ":TRIGger:EDGE:SOURce EXTernal")
    scopeCommand(scope, ":TRIGger:EDGE:SLOPe POSitive")
    scopeCommand(scope, ":TRIGger:EDGE:LEVel 1")

    # configure axes
    scopeCommand(scope, ":CHANnel1:COUPling AC")
    scopeCommand(scope, ":CHANnel1:SCALe 0.1")
    scopeCommand(scope, ":TIMebase:SCALe 0.00000056")
    scopeCommand(scope, ":TIMebase:POSition 0.0000025")

if __name__ == "__main__":
    print("--------------------------------")
    print("| SHA-3 Power Trace Capture |")
    print("--------------------------------")

    # open serial port to FPGA
    print("Opening connection to FPGA")
    fpga = getFPGA()

    # open connection to database
    print("Opening connection to database")
    conn = psycopg2.connect(dbname=DBNAME, user=DBUSER, password=DBPASS, host=DBHOST)
    cur = conn.cursor()

    # open and reset oscilloscope
    print("Opening connection to oscilloscope")
    scope = getScope()

    # configure oscilloscope
    print("Configuring scope")
    configCapture(scope)

    traces = 0

    # main loop
    while True:

        # generate input vector
        input_vector = b''
        if ENABLE_STATIC:
            input_vector = STATIC_INPUT
        else:
            input_vector = os.urandom(126)

        print("Generated input vector:")
        print(input_vector)

        # compute valid hash output
        expected_output = codecs.encode(keccak.hash(input_vector), 'hex_codec')

        print("Expected hash output:")
        print(expected_output)

        print("Setting Single Trace Capture")
```

```
scope.write(":SINGle")

sleep(0.5)

# pad input vector
input_vector += b'\x01\x80'

# send input vector to FPGA
print("Sending input vector to FPGA")
fpga.write(b'\x00\x01') # single test vector
fpga.write(rearrangeBytes(input_vector))

# wait for FPGA
while(fpga.in_waiting == 0):
    sleep(0.1)

# receive FPGA hash output
fpga_output = rearrangeHex(codecs.encode(fpga.read(32), 'hex_codec'))

print("Received FPGA hash output:")
print(fpga_output)

# compare hashes
if(expected_output == fpga_output):
    print("SUCCESS: FPGA hash matches expected value")

    # pull trace from oscilloscope
    trace = getTrace(scope)

    # compress trace
    print("Compressing power trace for storage")
    lz = lzma.LZMACompressor()
    comp_trace = lz.compress(trace)
    comp_trace += lz.flush()

    print("Trace compressed with ratio: " + str(len(trace) / len(comp_trace)))

    # write row to database
    print("Writing power trace to database")
    cur.execute('''INSERT INTO traces (config_id, capture_time, input_vector, compressed_trace)
                   VALUES (%(config_id)s, now(), %(input_vector)s, %(comp_trace)s)''',
                   { 'config_id': CONFIG_NUMBER,
                     'input_vector': input_vector,
                     'comp_trace': comp_trace })
    conn.commit()

    traces += 1
    print("Captured " + str(traces) + " traces")

else:
    print("FAILURE: FPGA hash does not match expected value")

if traces == CAPTURE_TRACE_COUNT:
    break
```

```
# close database, serial port, and oscilloscope connections
cur.close()
conn.close()
fpga.close()
scope.close()
```