Power Analysis Attacks on Keccak

Xuan D. Tran

Power Analysis Attacks on Keccak

by

Xuan D. Tran

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science
in Computer Engineering

Supervised by

Dr. Marcin Łukowiak and Dr. Stanisław P. Radziszowski
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
June 2015

Approved by:

Dr. Marcin Łukowiak, Associate Professor
Thesis Advisor, Department of Computer Engineering

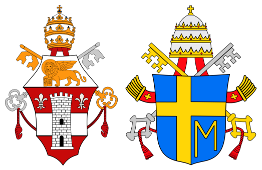
Dr. Stanisław P. Radziszowski, Professor
Committee Member, Department of Computer Science

Dr. Dhireesha Kudithipudi, Associate Professor
Committee Member, Department of Computer Engineering

Dedication

✠ Iesu Confido In Te ✠

Obedientia et Pax ✠ Totus Tuus



And He said, “What is impossible for human beings is possible for God.” (Luke 18:27)

To my mom, for her constant love and support.

Acknowledgments

I would like to express my deepest appreciation to all countless faculty, staff and friends, who provided me the possibility to complete this work. A special gratitude I give to my thesis advisors, Dr. Marcin Łukowiak and Dr. Stanisław P. Radziszowski for their continual guidance, suggestion and help to coordinate this work. Thank you for your patience to endure me for four long years. I would like to thank Dr. Dhireesha Kudithipudi for taking time out of her busy schedule to serve as a committee member. A special thanks goes to Dr. Manuel Lopez, who helped me with certain advanced mathematical concepts in some of my grad classes that were not only useful for my thesis but also for work-related. Furthermore, I would also like to acknowledge with much appreciation the crucial role of Nidhin Pattaniyil for algorithm consultation, James Thesing for Synopsys power trace discussion and guidance and, finally, James Evanko for reviewing and commenting on this thesis document.

Abstract

Power Analysis Attacks on Keccak

Xuan D. Tran

Supervising Professors: Dr. Marcin Łukowiak and Dr. Stanisław P. Radziszowski

Cryptographic hash functions are used in everyday applications from ensuring data integrity, to authentication and digital signatures. The current secure hash standards (SHS) defined by the National Institute of Standards and Technology (NIST) are: SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512. Breaking MD5, threats to SHA-1 and the emergence of new cryptanalysis techniques, led NIST to call for the creation of a new hash function, namely SHA-3, that will replace (or complement) the current SHS. There were 64 submissions initially in 2007 and by the last round, there were only five candidates left. On October 2, 2012, NIST announced Keccak as the winner of the SHA-3 competition. On May 28, 2014, NIST announced the publication of ”Draft FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions” with a 90-day public comment period. As of today, the SHA-3 standardization document is waiting to be finalized and submitted to the Secretary of Commerce for approval.

Side Channel Attacks (SCA) exploit weaknesses in implementations of cryptographic functions resulting from unintended inputs and outputs such as operation timing, electromagnetic radiation, thermal/acoustic emanations and power consumption to break cryptographic systems with no known weaknesses in the algorithm's mathematical structure. Power Analysis Attack (PAA) is a type of SCA that exploits the relationship between the power consumption and secret key (secret part of input to some cryptographic process) information during the cryptographic device normal operation. PAA can be further divided into three categories: Simple Power Analysis (SPA), Differential Power Analysis (DPA) and Correlation Power Analysis (CPA). PAA was first introduced in 1998 and mostly focused on symmetric-key block cipher Data Encryption Standard (DES). Most recently this technique has been applied to cryptographic hash functions.

Keccak is built on sponge construction, and it provides a new Message Authentication Code (MAC) function called MAC-Keccak. The focus of this thesis is to apply the power analysis attacks that use CPA technique to extract the key from the MAC-Keccak. So far there are attacks of physical hardware implementations of MAC-Keccak using FPGA development boards, but there has been no side channel vulnerability assessment of the hardware implementations using simulated power consumption waveforms. Compared to physical power extraction, circuit simulation significantly reduces the complexity of mounting a power attack, provides quicker feedback during the implementation/study of a cryptographic device and ultimately reduces the cost of testing and experimentation. An attack framework was developed and applied to the Keccak high-speed core hardware design from the SHA-3 competition using gate-level circuit simulation. The framework is written in a modular fashion to be flexible to attack both physical and power traces of AES, MAC-Keccak and future crypto systems. The Keccak hardware design is synthesized with the Synopsys 130-nm CMOS standard cell library. Simulated instantaneous power consumption waveforms are generated with Synopsys PrimeTime PX. 1-bit, 2-bit, 4-bit, 8-bit, and 16-bit CPA selection function key guess size attacks are performed on the waveforms to compare/analyze the optimization and computation effort/performance of successful key extraction on MAC-Keccak using 40 byte key size that fits the whole bottom plane of the 3D Keccak state. The research shows the larger the selection function key guess size used, the better the signal-to-noise-ratio (SNR), therefore requiring fewer numbers of traces needed to be applied to retrieve key but suffer from higher computation time. Compared to larger selection function key guess size, smaller key guess size has lower SNR that requires higher number of applied traces for successful key extraction and utilizes less computational time. The research also explores and analyzes the attempted method of attacking the second plane of the 3D Keccak state where the key expands beyond 40 bytes using the successful approach against the bottom plane.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Cryptographic Hash Functions

The purpose of a cryptographic hash function is to provide the assurance of data integrity. It is used in password authentication, digital signatures and other protocols. The hash function is used to construct a short digital fingerprint of some data also known as the message digest. Therefore, if the data is altered, the digital fingerprint is no longer valid. Regardless if the data is left in an insecure place, the integrity of the data can be checked from time to time by recomputing the fingerprint quickly and verifying it has not changed. An ideal cryptographic hash function is a one-way hash function that maps an arbitrary-length input message M to a fixed-length output hash $H(M)$

$$H : \{0,1\}^* \rightarrow \{0,1\}^m \tag{1.1}$$

such that the following properties hold:

- **One-way, or preimage resistance:** Given a hash $H(M)$, it is infeasible to find the message M.
- **Second preimage resistance, or weak collision resistance:** Given a message M_1, it is infeasible to find another message M_2 such that $H(M_1) = H(M_2)$.
- **Collision resistance, or strong collision resistance:** It is infeasible to find two messages M_1 and M_2 such that $H(M_1) = H(M_2)$.

Since the hash function can take an arbitrary-length input, it has to be fast, use little memory to produce the message digest, and be able to operate in stream mode.

There are two main usages of hash functions. The first category deals with taking an arbitrary-length input and then producing the message digest. An example of this would be the computer would hash the user input password to verify if it matches with the stored secret hashed value to authenticate the user's credentials. The second category is the keyed hash function often used in a message authentication code (MAC), where the key is, for example, prepended or appended to the data to be hashed. Since the data is hashed with the secret key, the message digest of the keyed hash function does not have to be securely stored away as with the previous category. The keyed hash function will be discussed in more detail later in the thesis.

1.2 SHA-3 Competition

There are many well known hash functions used by industries and government agencies. MD5 is a well known hash function that was designed by Ron Rivest in 1992. In the mid 1990s, NIST published SHA-0 and SHA-1, and then in 2001 published SHA-2 hash family functions: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256. The current Secure Hash Standard (SHS), that is a set of cryptographically secure hash algorithms (SHA) specified by NIST, specifies in document FIPS 180-4 the seven secure hash functions for government agencies to use comprised of SHA-1 and SHA-2 hash family function. With the practical and successful attacks on the MD5 hash function to find a collision in a few seconds and recent theoretical attacks on SHA-1 with 2^{63} operations versus brute force 2^{80} operations, NIST worries it is a matter of time before SHA-1 will be broken with the advent of faster supercomputers and better cryptanalysis techniques. Due to SHA-2 having similar algorithms as SHA-1, a successful attack on SHA-1 can also mean a security threat for SHA-2. This led NIST to call for the creation of a new standard cryptographic hash function, SHA-3, that will replace (or complement) the current SHS.

On November 2, 2007 NIST announced the public competition for the design of the SHA-3 hash algorithm in the Federal Register Notice. By October 31, 2008, NIST received 64 entries from cryptographers around the world and by December that year, it was narrowed down to 51 first round candidates. By the second round in July 2009, there were only

14 candidates, and only five finalists - BLAKE, Grøstl, JH, Keccak and Skein in December 2010 advanced to the third and final round [17]. The characteristics of the five remaining candidates regarding their designs, compression functions, and utilized cipher pieces are summarized in Table 1.1. For each round NIST hosted a conference to obtain public feedback. In addition, comments were sent to NIST and the public hash forum. Numerous papers were published in leading cryptographic journals and conferences regarding cryptanalysis and performance of remaining candidates. Finally after five years of carefully narrowing down the candidates, reviewing the public comments, and with final internal review, NIST announced Keccak as the winner of the SHA-3 cryptographic hash algorithm competition on October 2, 2012. On May 28, 2014, NIST announced the publication of "Draft FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions" with a 90-day public comment period. As of today, the SHA-3 standardization document is waiting to be finalized and submitted to the Secretary of Commerce for approval.

	BLAKE	**Grøstl**	**JH**	**Keccak**	**Skein**
Design:					
Merkle-Damgård	X	X			X
Sponge			X	X	
Compression Function:					
Not block cipher based		X			
Block cipher based, Davies-Meyer	X				
Block cipher based, Matyas-Meyer-Oseas					X
Used cipher pieces:					
AES		X			
ChaCha	X				
Threefish					X

Table 1.1: Characteristics of five finalists [1].

1.3 Contributions

The main contribution of this research is to explore and analyze the performance of power analysis attacks on hardware implementation of the Keccak hash function in a MAC-Keccak configuration, which utilizes one and two Keccak plane long keys, from generated simulated power traces using variable bit CPA attack selection function key guess sizes to extract the secret key. The work demonstrates the feasibility of applying the practical n-bit CPA attack against MAC-Keccak implementation using gate-level circuit simulated power consumption waveforms. An itemized list of specific contributions is listed here:

- Develop a standardized, easy to maintain and re-usable framework for performing simulated and physical trace power analysis attacks to facilitate the design and testing of crypto devices.
- Reproduce existing simulated power analysis attacks in [9] and [7] and physical power analysis attack in [12] on AES-128 to validate implemented framework.
- The first to apply side channel vulnerability assessment of the hardware implementation using gate-level circuit simulated power consumption waveforms from Synopsys EDA tools.
  - Develop, synthesize and simulate Keccak hardware implementation with a 130-nm CMOS standard cell library.
  - Create advantage over physical power extraction in terms of quicker feedback during the implementation/study of a cryptographic device and which ultimately reduces the cost of testing and experimentation.
- Investigation of 1-bit, 2-bit, 4-bit, 8-bit, and 16-bit CPA selection function key guess size attack performance:
  - Compare and analyze the optimization of n-bit CPA selection function key guess size and computation time performance of successful key extraction on MAC-Keccak that uses one Keccak plane (40 bytes) long key by analyzing signal-to-noise-ratio (SNR) and number of applied traces.

  - Explore and analyze method of attacking key expansion into the second plane of the 3D Keccak state.

The rest of the document is organized as follows: Chapter 2 discusses, in detail, the Keccak hash function, different side channel attacks and in depth discussions of the different types of power analysis attacks. The related works to this thesis are presented in Chapter 3. Chapter 4 discusses the Keccak high-speed core hardware design from the SHA-3 competition that is utilized in this thesis. Simulation and attack frameworks are discussed in Chapter 5. The results are discussed and analyzed in Chapter 6. Finally, the conclusion and some ideas for future work are presented in Chapter 7.

Chapter 2

Background

2.1 Keccak Hash Function

Keccak is a new hash function selected by NIST as the winner of the SHA-3 competition in 2012 to be the next SHA-3 standard. The current SHA-1 and SHA-2 secure hash standards defined in FIPS 180-4 were designed based on the Merkle-Damgård construction. As the underlying hash algorithm for SHA-3 and unlike its predecessors, Keccak was designed on the Sponge construction which means it could accept arbitrary length messages and generate digests of any desired sizes. The following are the main features of the Keccak function:

- The internal state-size for the Keccak-$p[b, n_r]$ permutation is comprised of b bits, where b is the fixed length of the strings that are permuted, called the width of the permutation and n_r is known as the number of rounds. The permutation is defined for any $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ and any positive integer n_r. The variable b can be considered to be a $5 \times 5 \times w$ array of bits, where $w = 2^l$, $b = 5 \times 5 \times 2^l$, and $l \in [0 : 6]$. The seven possible values for $b, w,$ and l that are defined for the Keccak permutations are noted in Table 2.1. By default, $l = 6$ and so the internal state has $b = 1600$ bits. A visualization of the parts of the state array for the Keccak permutation in terms of sheets, planes, slices, rows, columns, and lanes using the case $b = 200$ with $w = 8$ are illustrated in Figure 2.1. The messages are filled into the 3D array state from left to right beginning at the bottom plane and then proceeding into the upper planes.

b	25	50	100	200	400	800	1600
w	1	2	4	8	16	32	64
ℓ	0	1	2	3	4	5	6

Table 2.1: Keccak-p permutation widths and related quantities [2].

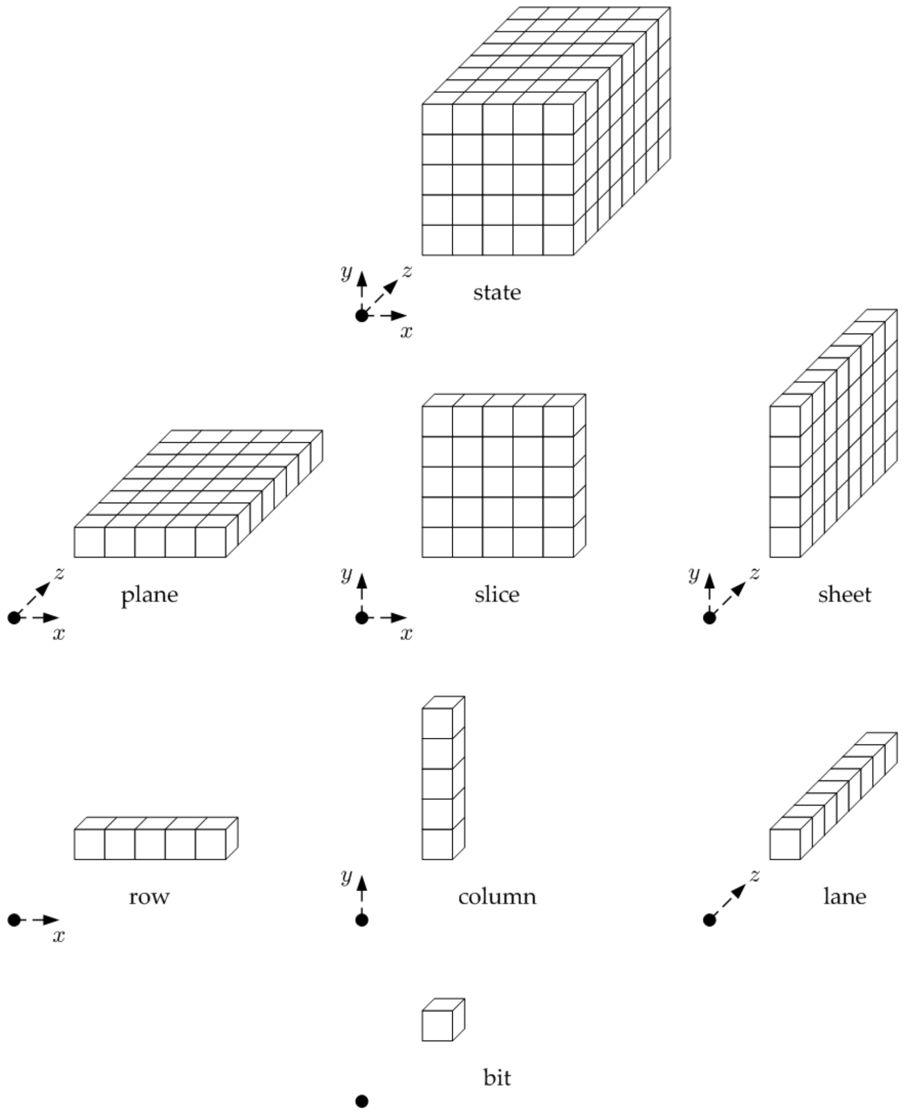


Figure 2.1: Parts of the state array, organized by dimension [2].

- The rate r defines the number of message bits that the sponge absorbs iteratively in every run, i.e. input block. Thus, the bit rate r determines the implementation speed.
- The capacity c is the number of zero bits that get appended to every r bits of the message to form the input state. Therefore, the state-size is the sum of the rate and capacity ($b = r + c$) as can be seen in Figure 2.2. The capacity c determines the security length such that the Keccak implementation with smaller r values absorbs more input blocks where each block has a smaller piece of information of the whole message to be hashed which is much more secure than implementation with larger r values. The tradeoff between smaller r with larger c values and larger r with smaller c values is security for speed, and vice versa.

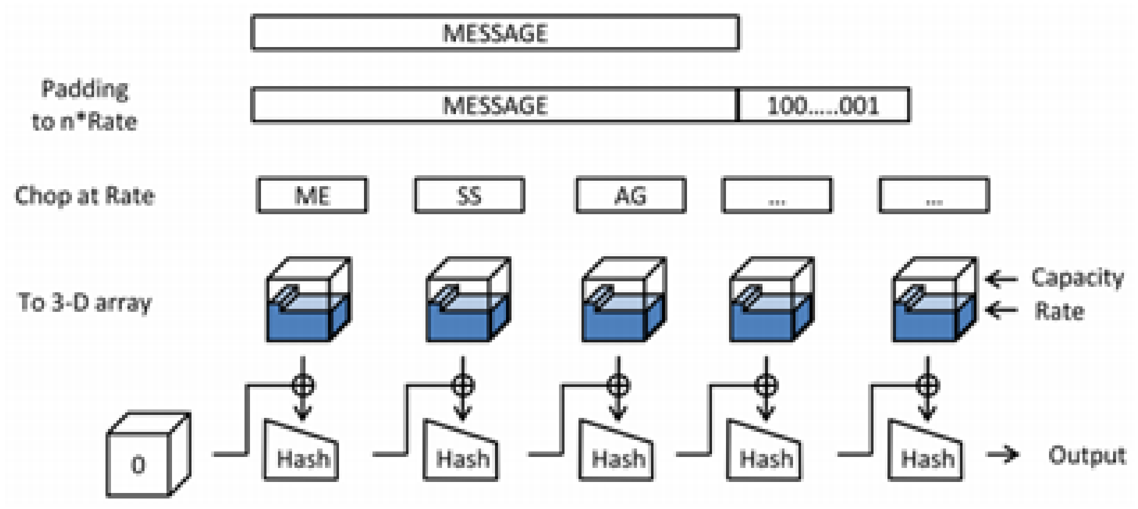


Figure 2.2: Keccak hash algorithm and the padding rules [4].

- Finally, the last configurable parameter of the Keccak function is the output length which is the size of the required digest in bits.

The rate, capacity and output length can vary depending on the security requirements. In the NIST round 3 submission, the designers proposed the rate r of 1152, 1088, 832, or 576 (144, 136, 104 and 72 bytes) for 224, 256, 384 and 512-bit hash sizes, respectively [2]. For the arbitrary length digest output, the default rate $r = 1024$, capacity $c = 576$ and the internal state size of $b = 1600$ bits. The Keccak function consists of $n_r = 12 + 2l$ rounds of five sequential transformations, where the number of rounds depends on the value l used to define the state size. A single round of Keccak is defined as

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta(Input). \tag{2.1}$$

The following sections will take a more detailed view of each transformation from the algorithmic perspective using pseudo-code. For more formal descriptions and mathematical explanations of each transformation, the reader is invited to read the Keccak reference specifications in [2]. The algorithm for each transformation takes a state array, denoted by $\mathbf{A}$, as an input and returns an updated state array, denoted by $\mathbf{A}'$, as the output [3]. The size of the state is omitted from the notation as it is understood b is always specified when the transformations are invoked.

2.1.1 Theta Transformation

The θ transformation is responsible for diffusion. It is a binary linear XOR operation with 11 inputs and a single output. As depicted in Figure 2.3, for every bit in the output state, it is the resulting XOR between itself and two intermediate bits produced by its two neighbor columns. The two intermediate bits are the resulting parity of the two columns from the XOR operation. The summation symbol, Σ, in Figure 2.3 denotes the parity from the XOR sum of all the bits in the column. The following shows the mathematic notation for the θ transformation along with the algorithm pseudo code implementation in Algorithm 1. From Algorithm 1 that is described in the Keccak reference document of [2], the efficient way to implement the θ transformation is to compute it over two successive phases. The first phase is to calculate the parity of each column, called θ_{plane}, which is Step 1 in Algorithm 1. The second phase consists of Steps 2 and 3 for calculating the remaining part of the θ transformation such that the two steps compute the XOR between every bit of the state and the two parity bits of θ_{plane}. By calculating the θ_{plane} first, it speeds up the calculation of the θ transformation versus the naive way of recalculating the parity bits of the two adjacent columns for every output bit.

$$\theta : \; a[x][y][z] \;\leftarrow\; a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1]$$

or equivalently

$$a[x][y][z] = a[x][y][z] \oplus (\oplus_{y'=0}^{4} a[x-1][y'][z]) \oplus (\oplus_{y'=0}^{4} a[x+1][y'][z-1])$$

Algorithm 1: θ(**A**) [2] [3]

Input: state array **A**

Output: state array **A**′

```
Steps:
    1. For all pairs $(x, y)$ such that $0 \leq x < 5$ and $0 \leq z < w$, let
           $C[x, z] = \mathbf{A}[x, 0, z] \oplus \mathbf{A}[x, 1, z] \oplus \mathbf{A}[x, 2, z] \oplus \mathbf{A}[x, 3, z] \oplus \mathbf{A}[x, 4, z]$.
    2. For all pairs $(x, y)$ such that $0 \leq x < 5$ and $0 \leq z < w$, let
           $D[x, z] = C[(x-1) \bmod 5, z] \oplus C[(x+1) \bmod 5, (z-1) \bmod w]$.
    3. For all triples $(x, y, z)$ such that $0 \leq x < 5$, $0 \leq y < 5$ and $0 \leq z < w$, let
           $\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z] \oplus D[x, z]$.
```

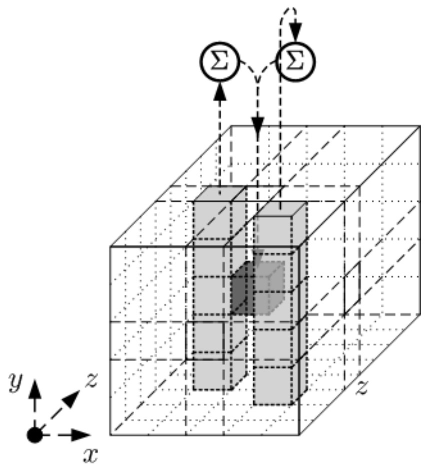


Figure 2.3: θ applied to a single bit [2].

2.1.2 Rho Transformation

The ρ transformation is a binary rotation over each lane of the state. It is the rotation of the bits of each lane by a specific length known as the offset which depends on the fixed x and y coordinates of the state. For each bit in the lane, the z coordinate is modified by adding the offset, modulo by the lane size w. The ρ transformation is a simple permutation over the bits of the state that is viewed as inter-slice dispersion. The following shows the mathematic notation for the ρ transformation along with the algorithm pseudo code implementation in Algorithm 2.

$$\begin{aligned} \rho:\ a[x][y][z] \ &\leftarrow\ a[x][y][z-(t+1)(t+2)/2], \\ &\text{with } t \text{ satisfying } 0 \le t < 24 \text{ and } \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ in GF}(5)^{2\times 2}, \\ &\text{or } t=-1 \text{ if } x=y=0 \end{aligned}$$

Algorithm 2: ρ(**A**) [2] [3]

Input: state array **A**

Output: state array **A′**

```
Steps:
    1.  For all $z$ such that $0 \le z < w$, let $\mathbf{A}'[0,0,z] = \mathbf{A}[0,0,z]$.
    2.  Let $(x,y) = (1,0)$.
    3.  For $t$ from 0 to 23:
        a.  For all $z$ such that $0 \le z < w$, let $\mathbf{A}'[x,y,z] = \mathbf{A}[x,y,(z-(t+1)(t+2)/2) \bmod w]$.
        b.  Let $(x,y) = (y,(2x+3y) \bmod 5)$.
    4.  Return $\mathbf{A}'$.
```

The calculation of the offsets for each lane can be seen from Step 3a in Algorithm 2. Using the internal state block size $b = 200$ bits with the lane size of $w = 8$ bits as an example, Table 2.2 lists the offsets for each lane that result from the computation in Step 3a. Figure 2.4 illustrates the effect of applying the offsets for each lane. Notice the labeling convention for the x and y coordinates from Figure 2.4 correspond to the rows and columns in Table 2.2 such that the lane $\mathbf{A}[0, 0]$ is in the middle of the sheet, and the lane $\mathbf{A}[2, 3]$ is bottom right of the sheet [3]. The black dot in each lane indicates the bit whose z coordinate is 0, and the shaded cube indicates the position of that bit after the execution of ρ. The other bits of the lane shift by the same offset, and the shift is circular resulting from the offset reduced modulo by the lane size [3].

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	153	231	3	10	171
$y = 1$	55	276	36	300	6
$y = 0$	28	91	0	1	190
$y = 4$	120	78	210	66	253
$y = 3$	21	136	105	45	15

Table 2.2: Offsets of ρ [2].

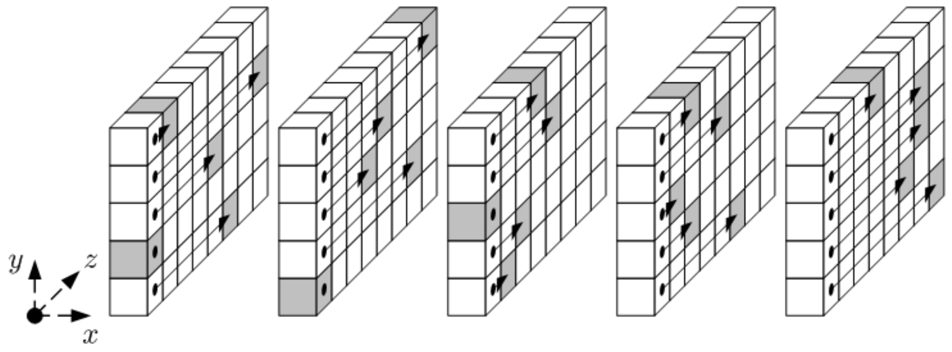


Figure 2.4: ρ applied to the lanes for $b = 200$. $x = y = 0$ is depicted at the center of the slices [2].

2.1.3 Pi Transformation

Similar to the ρ transformation as a simple permutation over the bits of the state, the π transformation is known for disturbing the slice horizontal and vertical alignment. The transformation shuffles every row of lanes to a corresponding column. The following shows the mathematical notation for the π transformation along with the algorithm pseudo code implementation in Algorithm 3. The π transformation rearranges the positions of the lanes for any slice as illustrated in Figure 2.5. The rearrangement of the positions of the lanes result from the computation of a mathematical equation in Step 1 of Algorithm 3. The convention for the labeling of the x and y coordinates is the same as that of the offset Table 2.2 of the ρ transformation, where the bit with coordinates $x = y = 0$ is depicted at the center of the slice.

$$\pi : \; a[x][y] \;\leftarrow\; a[x'][y'], \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$$

Algorithm 3: π(**A**) [2] [3]

Input: state array **A**

Output: state array **A**′

```
Steps:
    1. For all triples $(x, y, z)$ such that $0 \leq x < 5$, $0 \leq y < 5$, and $0 \leq z < w$, let
           $\mathbf{A}'[x, y, z] = \mathbf{A}[(x + 3y) \bmod 5, x, z]$.
    2. Return $\mathbf{A}'$.
```

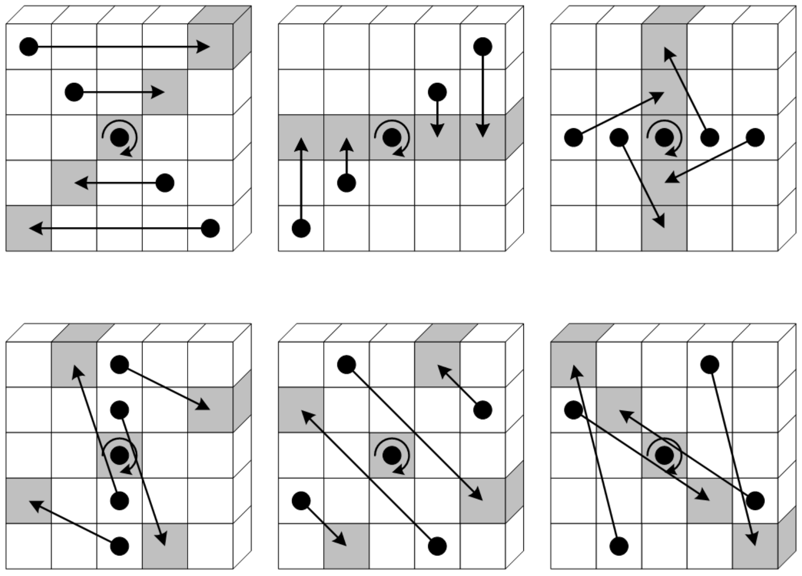


Figure 2.5: π applied to a single slice. $x = y = 0$ is depicted at the center of the slice [2].

2.1.4 Chi Transformation

The χ transformation is a non-linear operation that utilizes mixed XOR, AND, and NOT binary operations. Every bit of the output state is from the XOR between itself and the AND between one neighboring bit and the NOT of another neighboring bit in its row as illustrated in Figure 2.6. The output bit is flipped if its two adjacent bits along the x-axis are 0 and 1 in that order. The following shows the mathematical notation for the χ transformation along with the algorithm pseudo code implementation in Algorithm 4. From Step 1 of Algorithm 4, the dot on the right side of the equation indicates integer multiplication and in this case, is equivalent to the intended boolean AND operation [3]. Also from Step 1, the purpose of XORing with 1 is equivalent to a binary not operation.

$$\begin{aligned} \chi: \quad a[x] \quad &\leftarrow \quad a[x] + (a[x+1]+1)a[x+2], \text{ or} \\ a[x][y][z] &= a[x][y][z] \oplus (\overline{a[x+1][y][z]} \cdot a[x+2][y][z]) \end{aligned}$$

Algorithm 4: χ(**A**) [2] [3]

Input: state array **A**

Output: state array **A**$'$

```
Steps:
    1.  For all triples $(x, y, z)$ such that $0 \leq x < 5$, $0 \leq y < 5$, and $0 \leq z < w$, let
            $\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z] \oplus ((\mathbf{A}[(x+1) \bmod 5, y, z] \oplus 1) \cdot \mathbf{A}[(x+2) \bmod 5, y, z])$.
    2.  Return $\mathbf{A}'$.
```

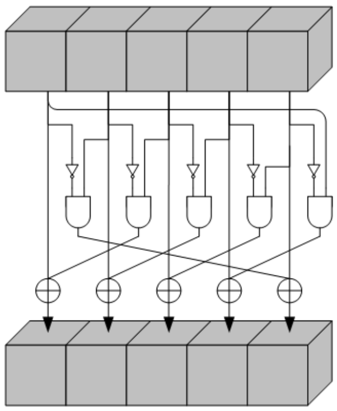


Figure 2.6: χ applied to a single row [2].

2.1.5 Iota Transformation

The ι transformation is a linear operation that XORs the bits of the first lane of the bottom plane (lane in origin) with a round-dependent constant that is associated with and depends on the round index, i_r, as seen below. The other 24 lanes are not affected by the ι transformation. Without ι transformation, the round mapping would be symmetric and all rounds would be the same.

$$\iota : \quad a \quad \leftarrow \quad a + \mathrm{RC}[i_r]$$

From the pseudo code of the ι transformation in Algorithm 6, the round constant denoted by RC is generated from the usage of the parameter i_r in Step 3 that determines $\ell + 1$ bits of a lane to be used for the round constant calculation. Each of these $\ell + 1$ bits is generated by a function denoted by rc in Algorithm 5, which is based on a linear feedback shift register (LFSR) [3].

Algorithm 5: $rc(t)$ [2] [3]

Input: integer t

Output: bit $rc(t)$

```
Steps:
    1. If $t$ mod 255 = 0, return 1.
    2. Let $R = 10000000$.
    3. For $i$ from 1 to $t$ mod 255, let:
        a. $R = 0||R$;
        b. $R[0] = R[0] + R[8]$;
        c. $R[4] = R[4] + R[8]$;
        d. $R[5] = R[5] + R[8]$;
        e. $R[6] = R[6] + R[8]$;
        f. $R = Trunc_8[R]$.
    4. Return $R[0]$.
```

Algorithm 6: $\iota(\mathbf{A}, i_r)$

Input: state array **A**, round index i_r

Output: state array **A**′

```
Steps:
    1. For all triples $(x, y, z)$ such that $0 \leq x < 5$, $0 \leq y < 5$, and $0 \leq z < w$,
        let $\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z]$.
    2. Let $RC = 0^w$.
    3. For $j$ from 0 to $\ell$, let $RC[2^j - 1] = rc(j + 7i_r)$.
    4. For all $z$ such that $0 \leq z < w$, let $\mathbf{A}' = [0, 0, z] = \mathbf{A}'[0, 0, z] \oplus RC[z]$.
    5. Return $\mathbf{A}'$.
```

The round constants RC$[i]$ can be implemented as a lookup table and are given in Table 2.3 below for the maximum lane size of $w = 64$ bits. For smaller sizes, they are simply truncated. The mathematical formula can be found in [2], and the pseudo code is in Step 3 of Algorithm 6.

RC[0] = 0x0000000000000001	RC[12] = 0x000000008000808B
RC[1] = 0x0000000000008082	RC[13] = 0x800000000000008B
RC[2] = 0x800000000000808A	RC[14] = 0x8000000000008089
RC[3] = 0x8000000080008000	RC[15] = 0x8000000000008003
RC[4] = 0x000000000000808B	RC[16] = 0x8000000000008002
RC[5] = 0x0000000080000001	RC[17] = 0x8000000000000080
RC[6] = 0x8000000080008081	RC[18] = 0x000000000000800A
RC[7] = 0x8000000000008009	RC[19] = 0x800000008000000A
RC[8] = 0x000000000000008A	RC[20] = 0x8000000080008081
RC[9] = 0x0000000000000088	RC[21] = 0x8000000000008080
RC[10] = 0x0000000080008009	RC[22] = 0x0000000080000001
RC[11] = 0x000000008000000A	RC[23] = 0x8000000080008008

Table 2.3: The round constants RC[i_r] [2].

2.1.6 Keccak and Sponge Construction

As discussed earlier, the Keccak-$p[b, n_r]$ permutation consists of n_r iterations of Rnd, as specified in Algorithm 7, where given a state array **A** and a round index i_r, the round function Rnd is the transformation that results from applying the operations θ, ρ, π, χ and ι, in that order, as illustrated below:

$$\text{Rnd}(\mathbf{A}, i_r) = \iota(\chi(\pi(\rho(\pi(\mathbf{A})))), i_r).$$

Algorithm 7: Keccak-$p[b, n_r](S)$ [3]

Input: string S of length b, number of rounds n_r

Output: string S' of length b

Steps:

1. Convert S into a state array, **A**.
2. For i_r from $2\ell + 12 - n_r$ to $2\ell + 12 - 1$, let **A** = Rnd(**A**, i_r).
3. Convert **A** into a string S' of length b.
4. Return S'.

As defined in the Keccak reference of [2], the special case of the Keccak-p family where $n_r = 12 + 2\ell$ is known as the Keccak-f family of permutations and is precisely defined in Equation (2.2). Therefore, the six SHA-3 functions that use the underlying Keccak-$f[1600]$

permutation are equivalent to the Keccak-$p[1600, 24]$ permutation.

$$\begin{aligned}\text{Keccak-}f[b] &= \text{Keccak-}p[b, n_r] \\ &= \text{Keccak-}p[b, 12 + 2\ell]\end{aligned} \tag{2.2}$$

The rounds of Keccak-$f[b]$ are indexed from 0 to $11 + 2\ell$. A result of the indexing within Step 2 of Algorithm 7 is that the rounds of Keccak-$p[b, n_r]$ match the last rounds of Keccak-$f[b]$, or vice versa. As an example, Keccak-$p[1600, 19]$ is equivalent to the last 19 rounds of Keccak-$f[1600]$. Also, Keccak-$f[1600]$ is equivalent to the last 24 rounds of Keccak-$p[1600, 30]$; in this case, the preceding rounds for Keccak-$p[1600, 30]$ are indexed by the integers from -6 to -1 [3].

Thus far, the discussion has been about the parameters and five operations of a Keccak single round at the low level. From the high level, the hash function operates in three steps: from initializing the hash function, to reading the arbitrary input (i.e. absorbing) and then producing the digest (i.e. squeezing), as can be seen in Figure 2.7. Due to the absorbing and squeezing steps, Keccak is a family of functions of sponge construction. A closer look at the sponge construction shows that it consists of the following three components: an underlying function of fixed-length strings denoted by f, a parameter called the rate denoted by r and a padding rule denoted by pad [3]. From these three components, Algorithm 8 is the pseudo code of the Sponge$[f, \text{pad}, \text{r}]$ function where M is the input message to the sponge function, d is the desired length of the output in bits and the width b is determined by the choice of f [3]. The following will discuss the three high level steps of a hash function using a sponge construction.

- **Initialization and Padding**

  Initialization of the function is achieved by setting the initial state to all zeros. Then, follow by padding the input message which appends a number of bits such that the total length is a multiple of the rate. The mandatory padding starts with '1', adds 0's in between, and ends with '1'; therefore, the minimum pad is 2 and the maximum length is $(r + 1)$. The pseudo code of the padding rule for the Keccak functions can be found in Algorithm 9. Note that * in the function name "pad10*1" of Algorithm

9 indicates the 0 bit is either omitted or repeated as necessary in order to produce an output string of the desired length.

- **Absorbing**

  In an analogy of a sponge, the function absorbs at a positive integer rate of r bits and is appended with a positive number of capacity bits, denoted by c, of zeros to fit the state size perfectly. Thus, the state consists of $b = r + c$ bits. The bits in the state are arranged in the 3D array ($5 \times 5 \times 2^l$) starting from $x = 0, y = 0, z = 0$. Essentially, the bits fill the bottom plane of the 3D array first, then move up the array in the z direction, then x direction and then y direction. The input of the hash function is the XOR of the new read in 3D message array with the previous state. This absorbing process continues for every block of message bits until there is no input message to process.

- **Squeezing**

  Finally, the function squeezes the first d bits, denoted by Trunc$_d$, of the output state, and this is the number of digest bits defined by the security requirements. The output can be regarded as an infinite string whose computation, in practice, is halted after the desired number of output bits is produced [3].

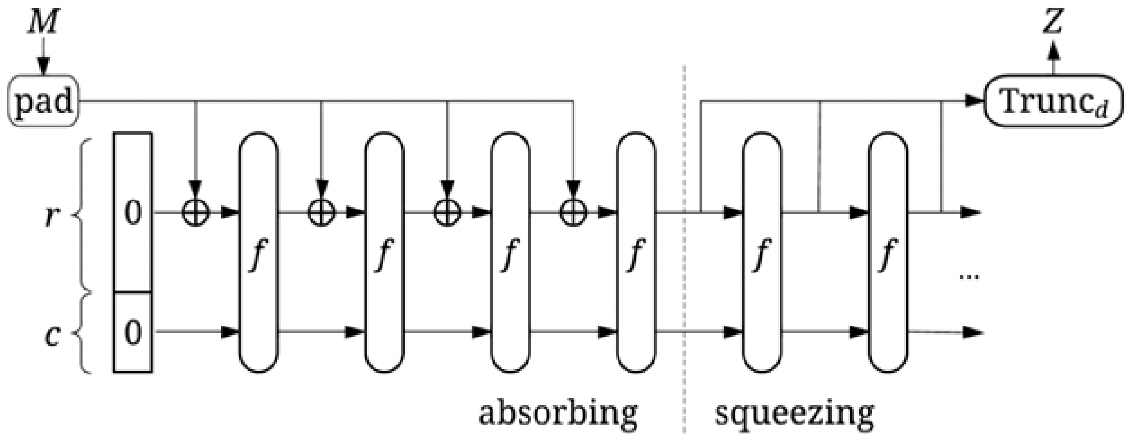


Figure 2.7: The sponge construction: $Z =$ Sponge[f, pad, r](M, d) [5][3].

Algorithm 8: Sponge[f, pad, r](M, d) [2] [3]

Input: string M, non-negative integer d

Output: string Z such that len$(Z) = d$

```
Steps:
    1. Let $P = M||\text{pad}(r, \text{len}(M))$.
    2. Let $n = \text{len}(P)/r$.
    3. Let $c = b - r$.
    4. Let $P_0, ..., P_{n-1}$ be the unique sequence of strings of length $r$ such that $P = P_0||...||P_{n-1}$.
    5. Let $S = 0^b$.
    6. For $i$ from 0 to $n - 1$, let $S = f(S \oplus (P_i||0^c))$.
    7. Let $Z$ be the empty string.
    8. Let $Z = Z||\text{Trunc}_r(S)$.
    9. If $d \leq |Z|$, then return $\text{Trunc}_d(Z)$; else continue.
   10. Let $S = f(S)$, and continue with Step 8.
```

Algorithm 9: pad10*1(x, m) [2] [3]

Input: positive integer x, non-negative integer m

Output: string z such that $m +$ len(Z) is a positive multiple of x

```
Steps:
    1. Let $j = (-m - 2) \bmod x$.
    2. Return $1||0^j||1$.
```

The standardization of the SHA-3 functions requires the internal state of the Keccak permutation to be 1600 bits; therefore, in the restricted case of $b = 1600$, the Keccak family is denoted by Keccak$[c]$. The SHA-3 function names are nothing more than aliases for different usage instances of Keccak$[c]$, such as SHA3-512(M) = Keccak$[1024]$ $(M||01, 512)$ where $c = 1024$. The choice of c, in this case, determines the r value. The technical expression of the Keccak$[c]$ in terms of the sponge construction using specific Keccak-p permutation with parameters, M, message and, d, output length is defined in Equation

(2.3).

$$\text{Keccak}[c](M,d) = \text{Sponge}[\text{Keccak-}p[1600,24], \text{pad10*1}, 1600-c](M,d) \tag{2.3}$$

2.1.7 FIPS 202 "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions"

This standard specifies the Secure Hash Algorithm-3 (SHA-3) family of functions on binary data. Each of the SHA-3 functions is based on an instance of the Keccak algorithm that NIST selected as the winner of the SHA-3 Cryptographic Hash Algorithm Competition. This standard also specifies the Keccak-p family of mathematical permutations (including the permutation that underlies Keccak) which can serve as the main components of additional cryptographic functions that may be specified in the future [3].

The Keccak-p permutations were designed to be suitable main components for a variety of cryptographic functions, including keyed functions for authentication and/or encryption. The SHA-3 family consists of four cryptographic hash functions called SHA3-224, SHA3-256, SHA3-384, and SHA3-512, and two extendable-output functions (XOFs) called SHAKE128 and SHAKE256 as can be seen below where these six functions share the same sponge construction and are known as sponge functions. The six SHA-3 functions use the same permutation as the main component in the sponge construction and, therefore, can be considered as modes of operation of the Keccak-$p[1600, 24]$ permutation [3]. The permutation operates on a Keccak state of 1600 bits with 24 rounds where each round consists of a sequence of five transformations: θ, ρ, π, χ and ι as defined in the previous sections.

SHA-3 Hash Functions

SHA3-224(M) = Keccak[448]($M||01, 224$)

SHA3-256(M) = Keccak[512]($M||01, 256$)

SHA3-384(M) = Keccak[768]($M||01, 384$)

SHA3-512(M) = Keccak[1024]($M||01, 512$)

SHA-3 Extendable-Output Functions

RawSHAKE128(M, d) = Keccak[256]($M||11, d$)

RawSHAKE256(M, d) = Keccak[512]($M||11, d$)

SHAKE128(M, d) = RawSHAKE128($M||11, d$) = Keccak[256]($M||1111, d$)

SHAKE256(M, d) = RawSHAKE256($M||11, d$) = Keccak[512]($M||1111, d$)

The numeric suffix of the four SHA-3 which hash functions indicates the fixed length of the digest such as SHA3-512 produces 512-bit digests. Due to the four SHA-3 hash functions having fundamentally different design principles, implementation and performance characteristics to the SHA-1 and SHA-2 families in FIPS 180-4 standard, they can be viewed as the supplement to the existing hash standard in FIPS 180-4. The advantage of the different design principles between SHA-3 and the hashes in FIPS 180-4 provides resilience against future advances in hash function analysis. The four SHA-3 hash functions differ slightly from the instances of Keccak in the SHA-3 competition by the concatenation of the two additional bits at the end of the messages. The purpose of this is to differentiate the SHA-3 hash functions from the SHA-3 XOFs and future new variants of the SHA-3 functions. The SHA3 standard defines this as domain separation. Overall, the hash functions are defined from the Keccak[c] function by appending two bits, i.e. 01, to the message as mentioned before and by specifying the length of the output [3]. For example, Keccak[1024]($M||01, 512$) is the technical representation of SHA3-512 that has the capacity of 1024 bits with 01 appended to the messages, 24 rounds of Keccak where each round has five transformations and with 512-bit output digest. Notice, in each case, the capacity is double the digest length, i.e. $c = 2d$.

The extendable-output format (XOF) is a function on binary data in which the output can be extended to any desired length [3] [2]. The two SHA-3 XOFs, SHAKE128 and SHAKE256, are defined from two intermediate functions, RawSHAKE128($M||11, d$) and RawSHAKE256($M||11, d$), which are defined from the Keccak[c] function. SHAKE stands for Secure Hash Algorithm Keccak. The input message is denoted by M and the output length is denoted by d. The bits 11 are appended to the message to support domain separation and also for compatibility with the Sakura coding scheme that uses tree hashing to compute and update the digest in parallel, efficiently, for long messages [3]. The suffixes of SHAKE128 and SHAKE256 indicate the security strength of the extendable-output function that they can generally support. SHAKE128 and SHAKE256 are the first XOFs that NIST has standardized and the approved uses of XOFs will be specified in NIST Special Publications [3].

2.2 Security Analysis of SHA-3

This section is a brief and high level summary of the security of the SHA-3 family of hash and extendable output functions. Since each of the SHA-3 functions is based on an instance of the Keccak algorithm that uses the sponge construction, the SHA-3 family inherits the security properties of the Keccak design and the sponge construction. The detailed analysis of the security properties of Keccak can be found in [2] along with the security properties of sponge construction in [5]. The six SHA-3 functions are designed to provide special properties, such as resistance to preimage, second preimage, and collision attacks. Extra security considerations need to be focused on the XOFs as these generate closely related outputs. The level of resistance of these three types of attacks for SHA-1, SHA-2 and SHA-3 is summarized in Table 2.4. From the table, regarding the security strength against second preimage attacks on a message M, the function $L(M)$ is defined as $\lceil \log_2(\text{len}(M)/B) \rceil$, where B is the block length of the function, i.e. 512 bits for SHA-1, SHA-224, and SHA-256, and 1024 bits for SHA-512 [3].

As supplements to the SHA-2 functions, the four SHA-3 hash functions are designed to provide resistance against preimage, second preimage, and collision attacks which equals or exceeds the resistance that the corresponding SHA-2 functions provide. These SHA-3 functions are also designed to resist other attacks, such as length-extension attacks, that would be resisted by a random function of the same output length, providing security strength up to the hash function's output length in bits, when possible [3].

XOFs are a new kind of cryptographic primitive that allows the flexibility to produce outputs with any desired length. These functions have the potential for generating related

Function	Output Size	Security Strengths in Bits		
		Collision	Preimage	2nd Premimage
SHA-1	160	< 80	160	$160 - L(M)$
SHA-224	224	112	224	$\min(224, 256 - L(M))$
SHA-512/224	224	112	224	224
SHA-256	256	128	256	$256 - L(M)$
SHA-512/256	256	128	256	256
SHA-384	384	192	384	384
SHA-512	512	256	512	$512 - L(M)$
SHA3-224	224	112	224	224
SHA3-256	256	128	256	256
SHA3-384	384	192	384	384
SHA3-512	512	256	512	512
SHAKE128	d	$\min(\frac{d}{2}, 128)$	$\geq \min(d, 128)$	$\min(d, 128)$
SHAKE256	d	$\min(\frac{d}{2}, 256)$	$\geq \min(d, 256)$	$\min(d, 256)$

Table 2.4: Security strengths of SHA-1, SHA-2, and SHA-3 functions [3].

outputs that designers of security applications, protocols or systems may not expect of hash functions. From the design perspective, the output length for an XOF does not affect the bits that it produces such that the output length is not a necessary input to the function. The output of the XOFs can be seen as an infinite string, and it is up to the user to invoke the function to compute the desired number of initial bits of that string. XOFs act as a hash function when processing input and behave as a stream function when producing output. When an XOF uses the same common message to produce two different output lengths, the two outputs are closely related to one another such that the longer output is an extension of the shorter output. As an example, given any message M and any positive integers d and e, Trunc$_d$(SHAKE256$(M, d + e)$) is identical to SHAKE256(M, d), where Trunc$_d$ is the string comprised of the first d bits of the passed in string parameter. The characteristic of using a common message to produce similar output with XOFs would never be exhibited with SHA-1, SHA-2, or SHA-3 hash functions when the hash functions of each family share identical structure. Therefore, development of any application, protocol, or system using an XOF needs to keep in mind the important security consideration of the XOF function due to its closely related output generation. SHAKE128 and SHAKE256 of SHA-3 are designed to resist preimage, second preimage, collision attacks and other attacks that would be resisted by a random function of the requested output length up to the security strength of 128 and 256 bits respectively. A random function whose output length is d bits cannot provide more than d bits of security against preimage and second preimage attacks and $d/2$ bits of security against collision attacks. As noted in Table 2.4, SHAKE128 and SHAKE256 will provide less than 128 and 256 bits of security, respectively, when d is sufficiently small [3].

2.3 HMAC and MAC-Keccak

A widely known MAC construction is the keyed-hash message authentication code (HMAC) that uses the cryptographic hash function in combination with a secret key [18]. With a secure HMAC implementation, it be must computationally infeasible for an adversary to forge a valid HMAC without the secret key, nor should it be feasible for the attacker to retrieve any information about the key. The HMAC is used to verify both the origin and data integrity of a message. The popularity of HMAC usage is due to its provable security and efficiency, given the underlying hash function is secure [19]. The HMAC algorithm accepts an arbitrary length message M and a secret key K and produces a fixed-length digest as follows:

$$HMAC(K, M) = H((K \oplus opad) \parallel H((K \oplus ipad) \parallel M)) \tag{2.4}$$

where H is a cryptographic hash function, K is a secret key padded to right with extra zeros until it matches with the input block size of the hash function, or the hash of the original key if it's longer than the block size, M is the message to be authenticated, $ipad$ is the inner padding (0x3636...3636) and $opad$ is the outer padding (0x5c5c...5c5c), both having the length of one block [18]. HMAC can be seen as a hash within a hash. The inner hash function HMAC construction takes the key variant $(K \oplus ipad)$ that is the same size as the input block and prepends the message and hashes the resulting block. The outer hash function of the HMAC prepends the digest of the inner hash function with another key variant $(K \oplus opad)$ and produces the HMAC digest. The need of the output key padding in the outer hash is to eliminate some security vulnerabilities related to returning the hash state directly as the output. With the FIPS PUB 198 standardization usage of HMACs, there have been no successful attacks on HMAC-SHA1 because the outer application of the hash function masks the intermediate result of the internal hash. Unlike Merkle-Damgård construction, the output of the sponge constructions only reveal a small part of the final state. Hence, it is believed that Keccak is protected, by design, of such vulnerabilities. The designers of Keccak have recommended to use the direct MAC construction with the Keccak, i.e. MAC-Keccak, output length smaller than the capacity [5].

$$MAC(K, M) = H(K \parallel M) \tag{2.5}$$

The secret key used in an HMAC is a clear target for adversaries to attack through traditional mathematical analysis or exploitation of side channels. In the Previous Work section, there will be more in depth discussion of the work in [4] by Taha *et al.* on the use of SCA on MAC-Keccak to obtain the key and how key length plays a role.

2.4 Side-Channel Attacks

The classical cryptanalysis perception of the cryptosystem operates on a set of inputs to produce some output and other information about the key is not available as in Figure 2.8. With this simplified view, the cryptanalytic methods attempt to exploit the algorithm by identifying weaknesses in its mathematical structure such as with linear and differential cryptanalysis [20].

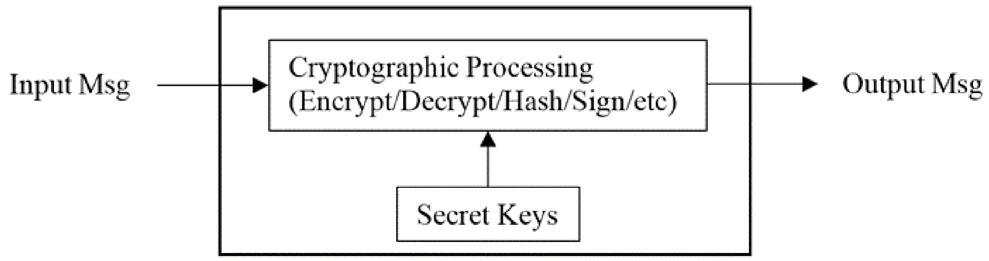


Figure 2.8: Traditional cryptographic assumptions [6].

When such a cryptographic algorithm is implemented on a physical device, it leaks some unintended information through what is known as side channels. There are several observable characteristics that can create side channels for potential information leakages such as power consumption, electromagnetic radiation, operation timing and others in Figure 2.9. Side Channel Attacks (SCA) have been developed to exploit these side channels on the physical cryptosystem implementation to extract secret information.

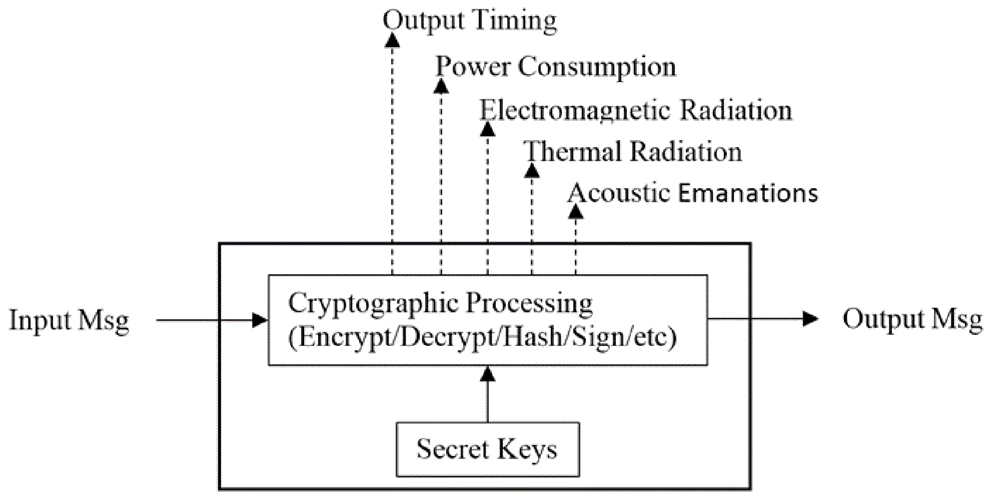


Figure 2.9: Information available to be exploited by SCA [6].

2.5 Power Analysis Attacks

One of the most powerful and best known side channel attacks is the Power Analysis Attack (PAA). It exploits the power consumptions of cryptographic devices to reveal the secret data used in cryptographic computations. When performing PAA, the attacker attempts to identify a relationship between the changing internal state of the cryptographic device and its measured power consumption. In order for this relation to be meaningful to the extraction of the secret data, an intermediate state must be identified that depends upon the secret data to be extracted [21] [7]. Power attacks represent a serious threat to unprotected cryptographic devices, given that they may be performed in a non-invasive manner using relatively cheap and easily obtained measurement equipment [6].

Source of Power Analysis

Prior to executing any power attack, the attacker needs to define a leakage model to predict the power consumption of the device under attack. In order to create an accurate model, one requires to know the technology used to synthesize a particular device and to understand how power is consumed in a typical circuit. Most modern cryptographic devices are implemented using complementary metal oxide semiconductor (CMOS) logic. The total power consumption of a CMOS circuit is the sum of the static power and dynamic power [8] as can be seen in Equation 2.6 and with an illustrated example of a CMOS inverter in Figure 2.10.

$$\begin{aligned} P_{total} &= P_{stat} + P_{dyn} \\ &= P_{stat} + (P_{tran} + P_{sc}) \end{aligned} \tag{2.6}$$

Static power consumption is the result of the device leakage current and the supply voltage. The dynamic power consumption occurs when the logic gates perform the output transitions. From a closer look at the dynamic power consumption, it is the combination of switching and short circuit power dissipation represented as P_{tran} and P_{sc}, respectively. The short circuit power is a result of the “short-circuit currents” when there exists a short period during the switching of a gate while PMOS and NMOS are conducting simultaneously. Dynamic power consumption is due to the charge and discharge of the load capacitance in the circuit. The importance of these dissipation sources typically depends on technology scalings. Since the dynamic power consumption is the dominating factor for the power consumption, it is particularly relevant from a side channel point of view because it determines a simple relationship between a device’s internal data and its externally observable power consumption [22]. The dynamic power consumption can be written as

$$P_{tran} = \alpha C_L V_{DD}^2 f \tag{2.7}$$

where α is the switching activity factor, C_L is the effective load capacitance, V_{DD} is the voltage of the power supply, and f is the clock frequency [8]. In CMOS devices, when measuring the power consumption either at the ground pin or at the power pin, the highest peak will appear during the charge of the capacitance, i.e. a $0 \rightarrow 1$ event [22]. This data-dependent power consumption is the origin of side-channel information leakages.

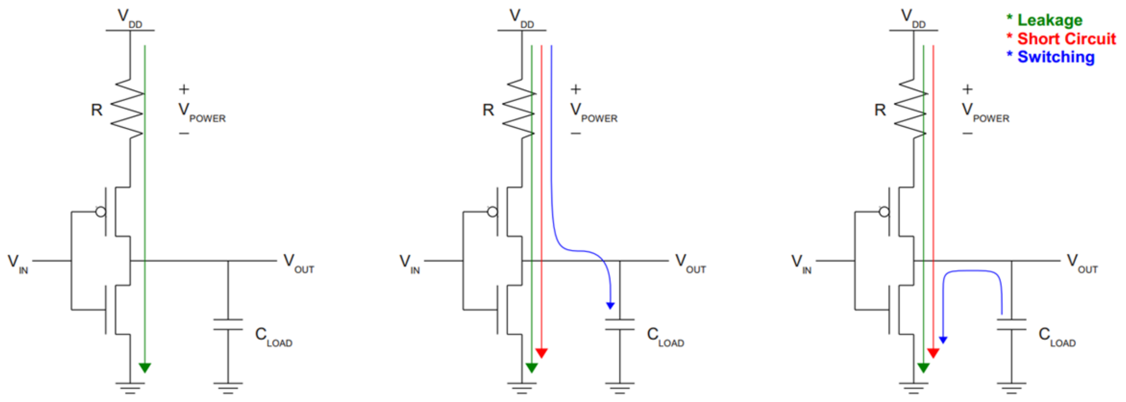


Figure 2.10: Idle consumption (left), charging (middle), discharging (right) [7] [8].

2.5.1 Power Leakage Models

To correlate the sensitive data with the power traces, there are different leakage models for the side-channel adversaries to use to predict the immediate power consumption of a device processing the target data. Some leakage models are more sophisticated than others. These power leakage models can be used both to simulate the attacks or to improve an attack's efficiency. The most common models used to describe the power leakage in CMOS devices are Hamming distance, Hamming weight and switching distance.

Hamming Weight Model

The Hamming weight model is the simplest representation of a device's power consumption such that the amount of power consumed is proportional to the number of bits that are logic '1' during an operation, namely $H_W(x_0)$. Therefore, the greater the number of bits that are set, the larger the amount of power consumed. This model is applicable for a microprocessor with precharged buses such as when the bus is set to zero between each value sent. Since the model is very simple, it weakly describes a circuit's power consumption but can be used as an advantage in cases where little is known about the underlying architecture [7].

Hamming Distance Model

This is the model that was commonly used to describe the power consumption in an embedded system. The model assumes that the number of bit transitions during a cryptographic operation is proportional to power consumption [23]. For example, when a value x_0 contained in the CMOS device switches into a value x_1, the actual side channel leakages are correlated with the Hamming distance of these values that be can expressed as $H_D(x_0, x_1) = H_w(x_0 \oplus x_1)$. In other words, it is the XOR of the two states such as that of a device register's past and present state, when using the Hamming weight model. If a bit is static during an operation, then it is assumed that it will not contribute to the power.

Switching Distance Model

Hamming weight and distance models assume $0 \rightarrow 1$ and $1 \rightarrow 0$ events consume the same amount of power. The switching model described by Peeters *et al.* [24] extends the Hamming distance concept by considering that the transition $0 \rightarrow 1$ may not consume the same amount of power as $1 \rightarrow 0$.

2.5.2 Attack Methodology

Power analysis attacks exploit a relationship between the changing internal state of a cryptographic implementation and the instantaneous power consumption to retrieve the secret key. Instantaneous power consumption defines the power consumption of an implementation (either hardware or simulation-based) that can be measured and recorded as it executes. The process of performing a successful power analysis attack presented by Smith in [9] is broken down into three important steps: identification, extraction and evaluation. Each step represents a part of the overall attack methodology that uses the instantaneous power consumption to identify the secret information.

Identification

In the identification step, the attacker identifies and hypothesizes a relationship between the secret key information and instantaneous power consumption. To have a meaningful relationship, the attacker needs to identify the secret data to attack and formulate an accurate power model that describes the circuit's power consumption by making certain assumptions about the device's architecture and implementation. Besides establishing a specific relationship between the secret key information and instantaneous power consumption, this step also includes identifying the required inputs to the system, the output values to be measured, and during which part of the execution the power consumption will be captured [9][7].

Extraction

The generation of the power waveforms may be either hardware or simulation-based; therefore, in the extraction step, a process is developed to collect and process the power measurements of the state of the relationship during execution and the necessary system inputs/outputs for an attack. The collection of measurements can be done in a non-invasive manner while the system performs a cryptographic operation. Since the attack may require a large number of power measurements, this step includes the development of an automated method that captures and stores the power traces, each with accompanying inputs and/or outputs and any other additional information [9][7].

Evaluation

The final step is to develop a software system to organize, process, and evaluate the collected power traces and system inputs/outputs. This information along with the leakage power model and attack algorithm are used to identify the relationship between the power trace data and estimated secret information to determine the most likely candidate for the target secret key information being sought. This step may be used to determine all or part of the secret key information [9][7].

2.5.3 Simple Power Analysis

Simple Power Analysis (SPA) is the most basic power analysis technique that attempts to interpret the power consumption of a device and deduce information about its performed operations [6]. The adversary observes a power waveform (or a number of waveforms) in order to identify large, noticeable features that may reveal information about the device's operations, or the data being operated on [7]. This concept is illustrated with the example in Figure 2.11 where the adversary could easily identify the Data Encryption Standard (DES) algorithm initial permutation, 16 noticeably repeated patterns also known as rounds and the final permutation at the end of the power trace. Since DES has 16 rounds and the visual inspection of the power trace confirms it, this is nothing new for the adversary, however, the adversary could use such a visual inspection of the leakage traces as the preliminary step in a more powerful attack by determining the parts of the traces that are relevant to the adversary [22]. There are cases in which this sequence of operations can provide useful information such as when the instruction flow depends on the data. An example is that SPA can be used to break Rivest Shamir Adleman (RSA) public-key decryption implementations by revealing differences between multiplication and squaring operations of the secret exponent's value. In the RSA algorithm, a simple modular exponentiation function scans across the exponent, performing a squaring operation in every iteration with an additional multiplication operation for each exponent bit that is equal to '1'. The secret exponent can be compromised if squaring and multiplication operations have different power consumption characteristics, take different amounts of time, or are separated by different code [6].

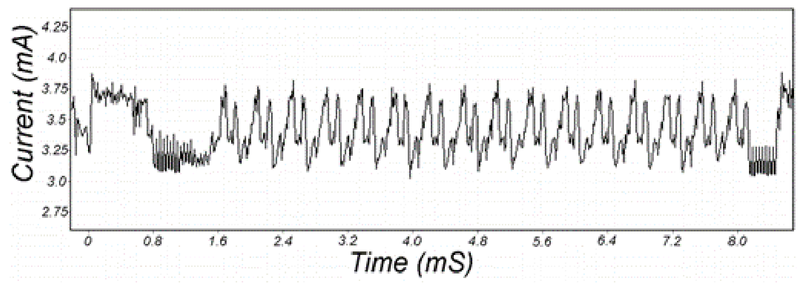


Figure 2.11: SPA trace showing DES algorithm's initial permutation, 16 rounds, and final permutation [6].

Since SPA is easy to launch, there are simple implementation techniques to prevent it. One technique is to avoid procedures that use secret intermediates or keys for conditional branching operations, which will mask many SPA characteristics. In cases where the algorithms inherently assume branching, this can require creative coding and incur a serious performance penalty [6].

2.5.4 Differential Power Analysis

Differential Power Analysis (DPA) is a powerful attack that is much more difficult to prevent than SPA. DPA intends to take advantage of data dependencies in the power consumption patterns such that the attacks use statistical analysis and error correction techniques to extract information correlated to secret keys from the detection of smaller scale variations that may otherwise be overshadowed by measurement errors or other noise when performing SPA [6].

Implementation of a DPA attack involves two phases: data collection and data analysis. The data collection phase may be performed in a non-invasive manner by sampling a device's power consumption during cryptographic operations as a function of time. The data analysis phase is to guess secret information by establishing a relationship between the secret information and the instantaneous power consumption of the device. The attacker would need to identify a state within the system that is dependent upon both the secret and known quantity, i.e. plaintext or ciphertext. The state is known as the sensitive value, and may be estimated by guessing the value of the secret given some known input. If the sensitive value is correlated to the circuit's power consumption, then a correct guess of the secret will correlate to the power consumption [6][7].

The DPA selection function $D(C, b, K_s)$ of a single bit DPA attack on a block cipher is defined as computing the value of the bit b of the sensitive state given a known ciphertext C (or plaintext) and a key guess K_s. The attacker observes m encryption operations and captures m power traces of k samples each, designated $\mathbf{T}_{1..m}[1..k]$. The attacker also records the corresponding ciphertexts $C_{1..m}$ (or plaintexts $P_{1..m}$).

DPA analysis uses power consumption measurements to determine whether a key block guess K_s is correct. To do this, the attacker computes a k-sample differential trace $\Delta_D[1..k]$ by finding the difference between the average of the traces for which $D(C, b, K_s)$ produces one and the average of the traces for which $D(C, b, K_s)$ produces zero as seen in Equation (2.8) [6]. Therefore, $\Delta_D[j]$ is the differential of the average power over $C_{1..m}$ of a given point j from the power consumption measurements.

$$\left.\begin{aligned} \Delta_D[j] &= \frac{\sum_{i=1}^{m} D(C_i, b, K_s)\mathbf{T}_i[j]}{\sum_{i=1}^{m} D(C_i, b, K_s)} - \frac{\sum_{i=1}^{m}(1 - D(C_i, b, K_s))\mathbf{T}_i[j]}{\sum_{i=1}^{m}(1 - D(C_i, b, K_s))} \\ &\approx 2\left(\frac{\sum_{i=1}^{m} D(C_i, b, K_s)\mathbf{T}_i[j]}{\sum_{i=1}^{m} D(C_i, b, K_s)} - \frac{\sum_{i=1}^{m} \mathbf{T}_i[j]}{m}\right) \end{aligned}\right\} \tag{2.8}$$

If K_s is incorrect, the correct value for bit b from evaluating $D(C, b, K_s)$ has the probability of $P \approx \frac{1}{2}$. In other words, the selection function's output is effectively uncorrelated to the target bit that was actually computed by the target device. The power traces will be divided into two subsets: a subset with traces for which $D(C, b, K_s)$ is one and another subset with traces for which $D(C, b, K_s)$ is zero, and the difference in the averages of the subsets should approach zero as the number of traces increases as in Equation (2.9) [6].

$$\lim_{m \to \infty} \Delta_D[j] \approx 0 \text{ if } K_s \text{ is incorrect} \tag{2.9}$$

If K_s is correct, the computed value of the selection function $D(C_i, b, K_s)$ produces the correct value of the target bit b with a probability of $P = 1$. The selection function is correlated to the value of the bit manipulated in the device target state. As a result, the $\Delta_D[j]$ approaches the effect of the target bit on the power consumption as the number of power traces increases. Any data values, measurement errors and anything else that are not correlated to D approach to zero. Due to the power consumption being correlated to the data bit values, the plot of Δ_D will have spikes in regions where D is correlated to the value being processed and flat or approaching zero elsewhere [6].

The DPA attack methodology just mentioned is to illustrate an attack of a single byte or word of the secret key from a cryptographic device. To recover the rest of the key bytes, the attacker would need to apply the same method until successfully retrieving the whole key. For example, AES-128 operates on a message block size of 128 bits and uses a 128 bit key. Since the AES state operates on 16 bytes and a key is 16 bytes, the attacker would need to find all 16 K_s that feed into 16 S-boxes of the first round encryption. The S-box is the target of interest because of its non-linear operation and high power consumption. Since the S-box takes 8 bit inputs and produces 8 bit outputs, the attacker would use $2^8 = 256$ key guesses as part of the selection function $D(C, b, K_s)$ and apply Equation (2.8). This is illustrated in Figure 2.12 using a simple block diagram. As an example, Figure 2.13 shows the case of when the DPA attack uses an incorrect key that results in a flat differential power trace. If the correct key guess is applied then there will be spikes in the different power traces as can been seen in Figure 2.14. There will be cases where some of the incorrect key guesses will produce different power traces with spikes that the attacker might conclude and mistake for the correct key. Thus, the attacker needs to examine the differential power traces of all the key guesses and looks for whichever key guess provides the highest spikes in the differential trace which is the likely correct key byte. The attacker would then reapply the same procedures to obtain the rest of the 15 key bytes of the AES-128 encryption. DPA can use known plaintext or known ciphertext and can find encryption or decryption keys such as with the AES algorithm.

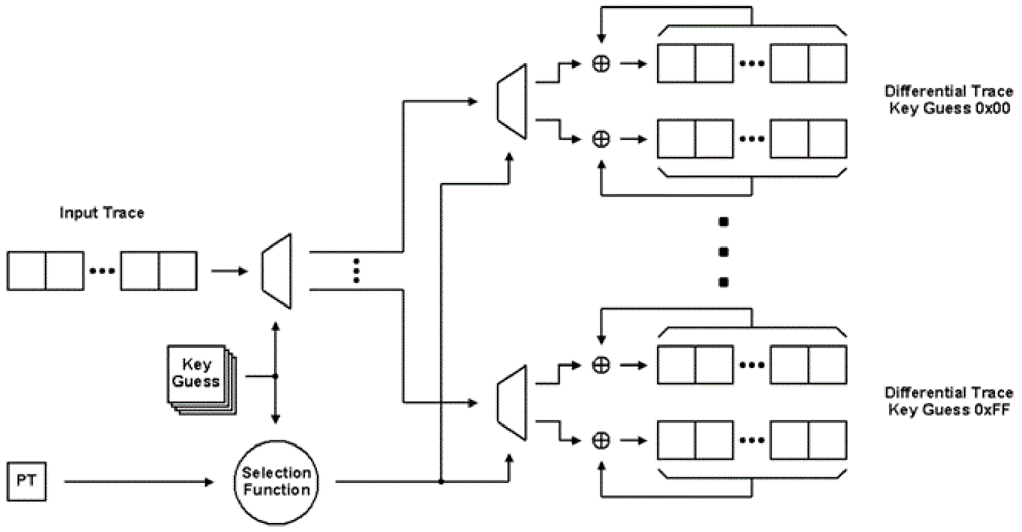

Input Trace
Key Guess
PT
Selection Function
Differential Trace Key Guess 0x00
Differential Trace Key Guess 0xFF

Figure 2.12: Differential power analysis overview [9].

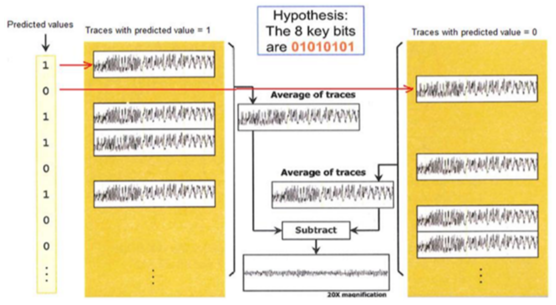

Predicted values
Traces with predicted value = 1
Hypothesis: The 8 key bits are 01010101
Traces with predicted value = 0
Average of traces
Average of traces
Subtract
20X magnification

Figure 2.13: DPA attacks with incorrect key guess [10].

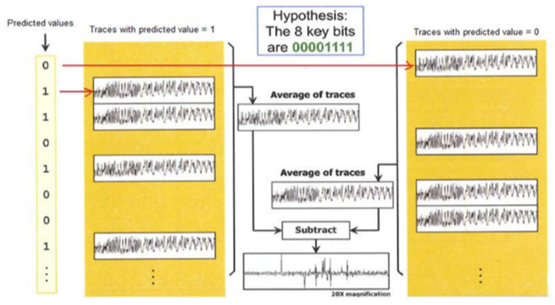

Predicted values
Traces with predicted value = 1
Hypothesis: The 8 key bits are 00001111
Traces with predicted value = 0
Average of traces
Average of traces
Subtract
20X magnification

Figure 2.14: DPA attacks with correct key guess [10].

The DPA algorithm may be extended to attack multiple bits at a time. The multi-bit attack computes an n-bit quantity and computes the average trace for the trace set bin 1 and trace set bin 0 on a specified threshold. The manner in which the traces are partitioned depends upon the leakage modes and the value of the threshold [7][9]. The advantage of the multi-bit attack over the single bit attack is in the improvement of signal-to-noise ratio (SNR) which leads to a smaller number of power traces to launch a successful attack [25].

Another type of DPA is the high-order DPA (HO-DPA) that uses sophisticated selection functions that combine multiple samples from within a trace. Selection functions can also assign different weights to different traces or divide traces into more than two categories. It is able to defeat many countermeasures or attack systems where partial or no information is available about plaintexts or ciphertexts to the attackers. Data analysis using functions other than ordinary averaging are beneficial with data sets that have unusual statistical distributions. Although HO-DPA is way more powerful than the simple DPA that uses ordinary averaging, it is much more complicated to implement [6].

2.5.5 Correlation Power Analysis

Different from DPA, Correlation Power Analysis (CPA) deduces the correct key by using correlation coefficients of statistics. This method was first introduced by Brier *et al.* on DES and AES [26]. In CPA, the attacker computes the correlation between two independent variables that are the sensitive value calculated by the power model, i.e. Hamming Weight or Hamming Distance, and the actual device's power consumption. The sensitive value is computed in the same way as DPA using a known input such as plaintext or ciphertext and key guess to estimate the intermediate state, and applying a power model to approximate its power contribution [21][7].

Let's denote by X the predicted power calculated by a power model and by Y the equivalent real power traces measured when processing the cryptographic operation. The Pearson's correlation coefficient $\rho_{X,Y}$ between X and Y with their expected values μ_X and μ_Y and standard deviations σ_X and σ_Y can be calculated as

$$corr(X,Y) = \rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \tag{2.10}$$

where E is the expected value function [27]. Since power analysis deals with discrete measured power traces, Equation (2.10) can be rewritten as

$$r_{x,y} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{2.11}$$

or as the mean of the products of the standard scores

$$r_{x,y} = \frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{s_x}\right)\left(\frac{y_i - \bar{y}}{s_y}\right) \tag{2.12}$$

and is referred to as the Pearson correlation coefficient r [27]. From the standard score expression in Equation (2.11), the sample mean $\bar{x}$ and sample standard deviation s_x can be calculated as

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \text{ and } s_x = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{2.13}$$

The sample Pearson correlation coefficient formula may be further rearranged to facilitate the construction of a single-pass algorithm for calculating sample correlations, but depending on the numbers involved, i.e. digit precision, it can sometimes be numerically unstable [27].

$$r_{x,y} = \frac{\sum x_i y_i - n\bar{x}\bar{y}}{(n-1)s_x s_y} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{(n-1)\sum x_i^2 - (\sum x_i)^2}\sqrt{(n-1)\sum y_i^2 - (\sum y_i)^2}} \tag{2.14}$$

The correlation coefficient indicates how well two random variables match each other. Therefore, the correlation coefficient r is a unitless quantity between -1 and 1, where -1 is strong negative correlation, 0 is no correlation, and 1 is strong positive correlation [27].

As mentioned earlier, in the CPA attack, the value of a secret key is hypothesized and then the power model, i.e. Hamming weight or Hamming distance, of some intermediate value is calculated. The higher the absolute value of $r_{x,y}$, the better correlation matches between the measured power consumption and the hypothetical power consumption, i.e. Hamming weight or Hamming distance [13]. The highest absolute value of $r_{x,y}$ suggests the correct hypothesized key.

Compared to DPA, CPA requires more intensive math calculations but due to its ability to pick up subtle characteristics of the power traces that DPA is not able to exploit, it requires less power traces to launch a successful attack. In general, DPA requires a large number of traces due to all the unpredicted data bits penalized by the signal-to-noise ratio (SNR) in the case of using smaller number of traces[26] [28]. Another approach to improve DPA with SNR issues is by using the DPA multi-bit attack [25][9].

Chapter 3

Previous Work

Initial Work

Power analysis attacks were first introduced by Paul Kocher, Joshua Jaffe, and Benjamin Jun from Cryptography Research Inc. in 1999 [6]. The paper describes how and why the power consumption of an implementation can be related to secret information. It defines simple, differential and high-order differential power analysis attacks and provides experimental results for single-bit DPA attacks against commercial smart-card devices using DES implementation. The authors describe the methodologies on how to attack the first and last rounds of DES to obtain the 56-bit key. The important notes from this paper are the definition of the DPA selection function $D(C, b, K_s)$ and the k-sample differential trace formula from Equation (2.8) that are essentially the backbone of the DPA algorithm.

Early Stage of DPA and CPA Attack Applications

A paper by Aigner and Oswald in [21] extends the methodology of a single-bit DPA attack by Kocher *et al.* [6], demonstrating such an attack on software implementation of DES. The authors present the methodology for developing a simulated power model and also describe the acquisition environment used to capture power measurements from an 8-bit Atmel microprocessor on an 8052-board.

The basic single-bit DPA attack was extended to multi-bit DPA attack by Messerges *et al.* in [23]. The goal is to increase the signal to noise ratio (SNR) of the differential trace for a correct key guess. With higher SNR, it requires fewer number of traces and shorter time to obtain a successful attack. The technique is applied to a software implementation of DES. A selection function is used as with single-bit DPA in order to separate the power traces and ciphertexts/plaintexts into groups. With multiple-bit DPA, the selection function is modified so that it outputs multiple bits. With these multiple output bits, the attack is able to partition the average traces of group “1” and group “0” according to a simple thresholding scheme, referred to as the “generalized d-bit DPA attack”. The variable d is a threshold of how many bits are “1” from the number of output bits in the selection function that is represented by “n”. The difference between group “1” and group “0” is the differential power trace for that particular key guess. In the case where the threshold is selected such that the output of the selection function is all 0’s or all 1’s, the attack is referred to as the “all-or-nothing d-bit DPA attack” [7]. In addition to the discussion of Hamming weight attacks, the paper also talks about the attacks based on Hamming distance.

Correlation power analysis attack was first introduced by Brier *et al.* in [26] on AES. The paper discusses the mathematics and the algorithm behind CPA attack. The attack relies on the Pearson correlation coefficient formula where the attacker computes the correlation between two independent variables that are the sensitive value calculated by the power model, i.e. Hamming Weight or Hamming Distance, and the actual devices power consumption. The authors demonstrated the attack on a large set of smart card chips using AES and the results showed the advantages of the CPA method against DPA in terms of efficiency and robustness such that it takes far fewer traces to successfully attack AES than DPA. The paper mentions the drawback of CPA which is the characterization of the leakage model parameters as it is more demanding than DPA, i.e. computational intensive; therefore the method may be more difficult to implement.

DPA and CPA Attacks on AES and Grøstl Utilizing ASIC Simulated Power Trace Framework

Messerges *et al.* in [23] presented the idea of the multi-bit DPA attack on DES as discussed earlier. Ken Smith in [9] extended the work of Messerges *et al.* by formalizing an attack methodology and creating a framework for performing single and mult-bit DPA attacks on AES hardware implementation. The attack methodology is divided into three stages:

identification, extraction and evaluation. Smith's implementation of the extraction stage is flexible for use on simulated power waveforms and also waveforms captured from physical implementation. An AES hardware is developed and synthesized with the Synopsys 130-nm CMOS standard cell library. The Synopsys PrimeTime PX tool is then used to generate simulated instantaneous power consumption waveforms for single and multi-bit DPA attacks. A set of scripts are developed to automate the process of simulating the design, capturing/storing the resultant power measurements, and invoking the attack framework to execute the user's choice of single or multi-bit DPA as depicted in Figure 3.1. Smith uses the known plaintext attack to feed plaintexts into the AES engine and then use the hamming weight of the byte substitution output of the AES first round as the sensitive data to successfully retrieve the keys from single and multi-bit DPA attacks.

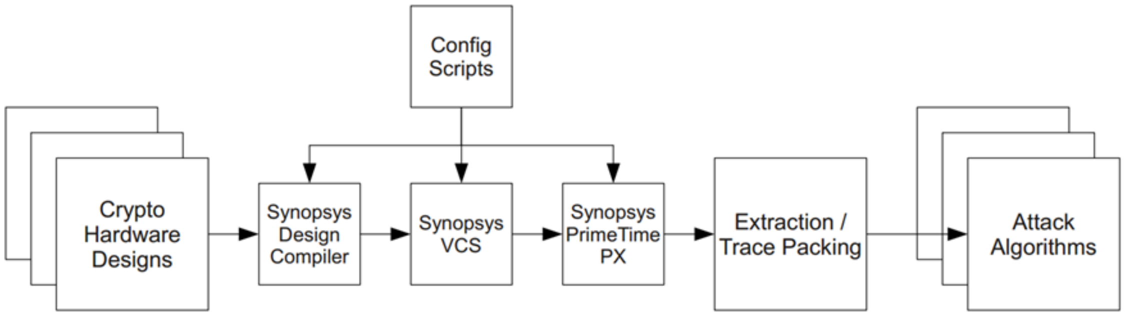


Figure 3.1: Ken Smith's thesis: ASIC top-level simulation flow [9].

The attack methodology by Ken Smith in [9] serves as the foundation for the work by Garrett Smith in [7] to perform power analysis attacks on the SHA-3 candidate Grøstl. Compared to Ken Smith's work, Garrett Smith makes significant improvement in the simulation flow and attack framework in order to synthesize, simulate and attack a given cryptographic algorithm. A number of alterations are made to the flow to reduce the simulation time. These include changes to the test bench structure, a careful selection of the simulation window, significant reduction in disk usage between each of the process, and most importantly, modification of the two most time consuming processes (simulation and power evaluation) to execute in parallel [7]. From Figure 3.2, the process is mostly the same as in Figure 3.1, but with the modification of having the simulation and power evaluation execute in parallel. Similar to the previous work, the implemented cryptosystem designs use the Synopsys 130nm standard cell library. From the attack framework perspective, it is written in a modular, plugin-oriented fashion to facilitate the additional support of new cryptosystems, leakage models and attack algorithms [7]. Smith demonstrates successful single- and multi-bit DPA and CPA attacks on AES. For Grøstl implementation with 64-bit data path, only single- and multiple-bit CPA attacks succeeded. Finally, only the multi-bit CPA attack is successful on the Grøstl implementation with 512-bit data path. Smith also implements countermeasures of the boolean masking technique on these designs to thwart the first order power analysis attacks.

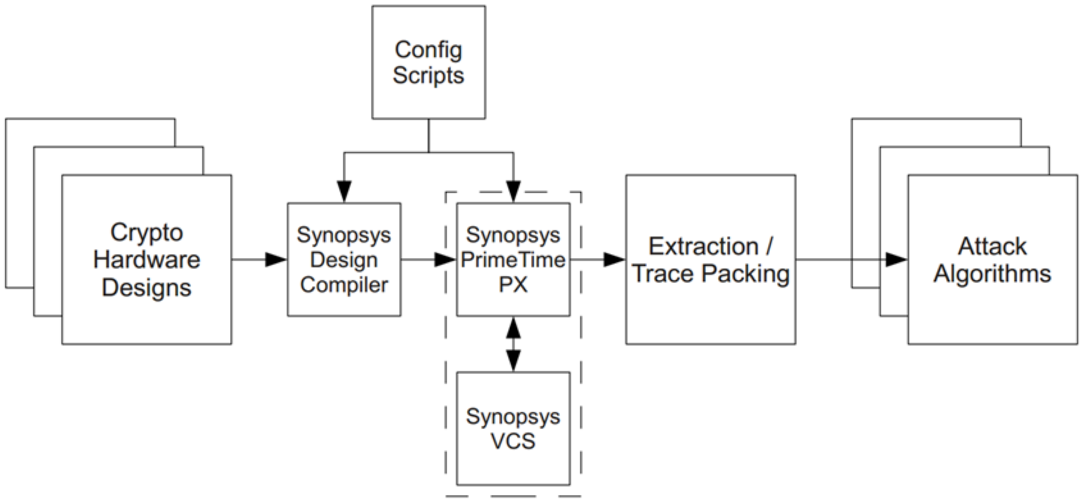


Figure 3.2: Garrett Smith's thesis: modified ASIC top-level simulation flow [7].

CPA Attacks on AES Implementation on FPGA Development Boards

The previous work contained theoretical simulated attacks on AES to retrieve the key such as in [9] and [7]. Keven Meritt [12] and Benhadjyoussef *et al.* [11] demonstrate the actual successful physical attacks on AES-128 on FPGA boards using CPA technique within a few minutes. Meritt uses the Xilinx Spartan-3E starter kit and Benhadjyoussef *et al.* use the Side-channel Attack Standard Evaluation Board (SASEBO) that uses a Xilinx FPGA. The SASEBO is designed specially to have mounted test points ready for side-channel attacks. In fact, the same board type is used by various organizations working on cryptographic implementations and side-channel attack research such as the well known dpacontest.org group. The board used by Meritt was not designed for SCA purposes, so Meritt makes the following hardware modification: removal of the decoupling capacitors for the FPGA core voltage to get the best quality power traces and addition of the shunt resistor in series with the FPGA V_{cc} power rail to measure the instantaneous power consumption during the encryption cycles. Both the work by Meritt and Benhadjyoussef *et al.* use the known ciphertext attack with CPA technique, in particular with the Hamming distance power model, to retrieve the round 10^{th} key as in Figure 3.3.

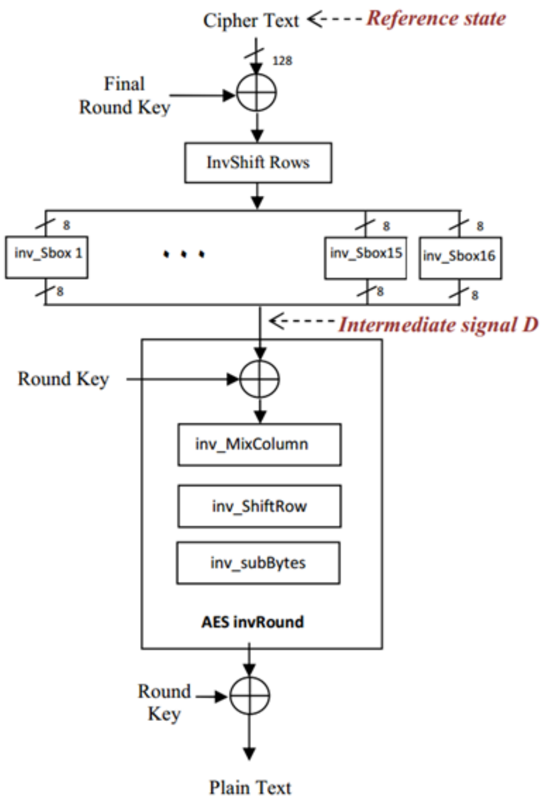


Figure 3.3: AES round 10 CPA attack using Hamming distance model [11].

The Hamming distance power model in this case is the number of bits that change from the reference state, i.e. ciphertext, to internal signal state D. By recovering the round 10^{th} key, it is a matter of performing the reverse key expansion to obtain the original key. Less than 10,000 actual power traces are needed to retrieve 16 key bytes which is in the same range for simulated power attacks by Ken Smith in [9] and Garrett Smith in [7]. The papers show the feasibility of attacking an AES implementation in an FPGA without requiring any expensive equipment, and once properly set up, are able to complete successful attacks in an order of minutes.

It is worthwhile to look a little deeper into Meritt's physical attack setup. The power traces generated from this setup serve as test traces for the test verification of the development of this thesis attack framework from reading traces, processing traces, applying different power analysis attacks, i.e. SPA, DPA, and CPA, to implementing different power models. Figure 3.4 shows the Xilinx Spartan-3E FPGA block diagram that uses the naive and unprotected implementation of AES-128. The overall interactions of the different components of the attack setup such as the PC, target FPGA development board and the oscilloscope can be seen in Figure 3.5. The heart of the CPA attack application from Meritt's work is the waveform acquisition program as illustrated in Figure 3.6 of the software class structure. It interacts with the FPGA development board to send down plaintexts and receive ciphertexts, interacts with the oscilloscope to collect power traces and finally, executes the CPA attack when finished collecting the ciphertexts and corresponding power traces.

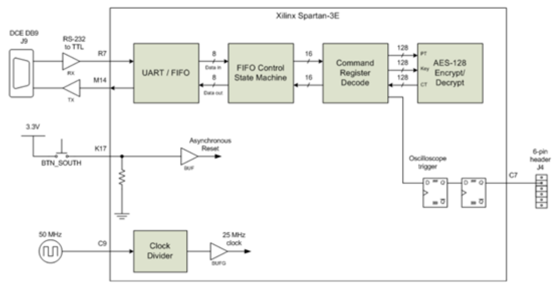


Figure 3.4: Xilinx Spartan-3E FPGA block diagram [12].

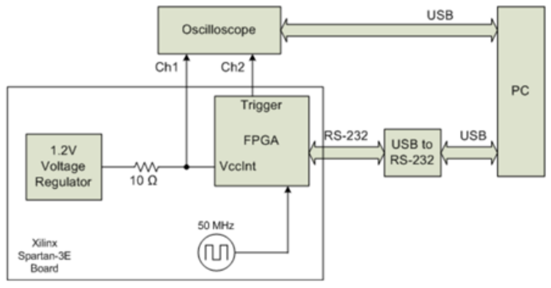


Figure 3.5: CPA platform setup using FPGA development board [12].

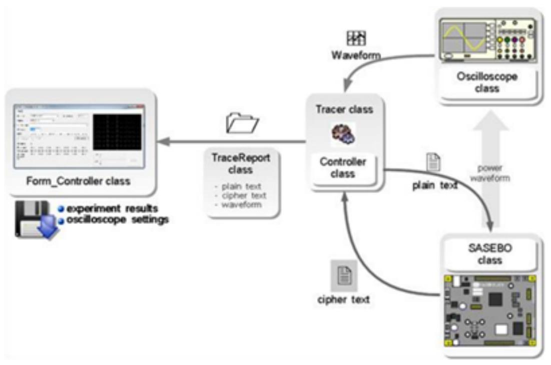


Figure 3.6: Waveform acquisition software class structure [12].

DPA and CPA Attacks on Existing Hash Functions in HMAC Construction with Simulated and Physical Traces

The previous research discussed thus far examined DPA and CPA attacks on simulated and actual captured power waveforms from block cipher encryptions of DES and AES. The usage of power analysis attacks is not limited to block cipher encryptions. There are a number of research papers which examine the usage of power analysis attacks on HMAC to retrieve the key or to obtain an internal state so the attacker could forge the message. Robert McEvoy *et al.* use DPA to attack a simplified version, SHA2-HMAC, that is only the inner hash function of the HMAC, specifically $H((K \oplus ipad) \parallel M))$ [29]. The purpose is to recover a fixed intermediate hash of $(K \oplus ipad)$ using practical DPA 1^{st} order attacks. The attack is not to recover the secret key, but the secret intermediate state of the hash function such that it allows an attacker to forge MACs for arbitrary messages. The paper demonstrates the implementation on the Xilinx Spartan-3E development board using SHA-256. The traces are obtained for the first three rounds of SHA-2, 64 rounds with 4,000 random messages, but execute 700 times per message to reduce acquisition noise for use in the analysis of seven variables of a SHA-2 state. The authors also discuss a countermeasure design using boolean masking for linear operations and arithmetic masking for non-linear operations on SHA-2 operations to protect against power analysis attacks. The paper talks about an algorithm for converting back and forth between the boolean and arithmetic maskings. The caveats with the proposed countermeasure are that more LUT (lookup table) resources from the FPGA will be used for masking operations and there will be degradation of hash performance due to converting back and forth between the boolean and arithmetic maskings.

Similar to the work by McEvoy *et al.* which uses power analysis to obtain the internal state of the HMAC to forge MACs on arbitrary messages, Fan Zhang and Zhijie Shi in [13] use DPA and CPA attacks on a fully implemented HMAC-Whirlpool to obtain the hashes of $H(K \oplus ipad)$ and $H(K \oplus opad)$ as the initialization inputs of the HMAC-Whirlpool to forge messages as illustrated in Figure 3.7. The motivation of the authors to pick Whirlpool as the underlying hash function for HMAC is because Whirlpool is a block cipher based hash algorithm using Miyaguchi-Preneel construction. It has been in the public domain more than a decade and so far no effective attacks have been found. Hence, using HMAC-Whirlpool is supposed to be secure. It is difficult to retrieve the key because the hash function has good one-way property, so an alternative is to find the intermediate hash value after the key is used. From Figure 3.7, the attacker obtains H_1' and H_1'' of the $H(K \oplus ipad)$ and $H(K \oplus opad)$ output respectively, then uses them as initialization inputs of a modified HMAC without needing to know the key and is, therefore, able to forge any given message successfully. Similar to past research where the S-Box was the primary target for the selection function due to its non-linear operation corresponding to high power consumption on physical devices, Zhang *et al.* also exploit the S-Box as the selection function for the power analysis attacks using a known plaintext attack approach as seen in Figure 3.8. The authors use DPA and CPA with the Hamming weight model to retrieve H_1' and H_1'' successfully, using less than 10,000 inputs on an 8-bit Atmel AVR processor running HMAC-Whirlpool. Furthermore, the paper confirms the use of CPA to retrieve H_1' and H_1'' with a smaller number of inputs than DPA, which is in line with past research for DPA and CPA attacks on AES.

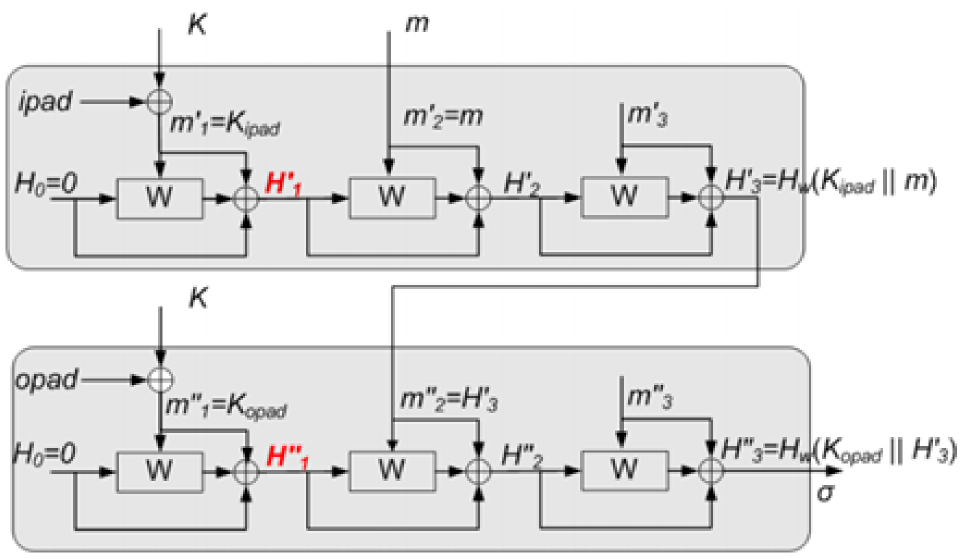


Figure 3.7: HMAC-Whirlpool where both K and m are one block long[13].

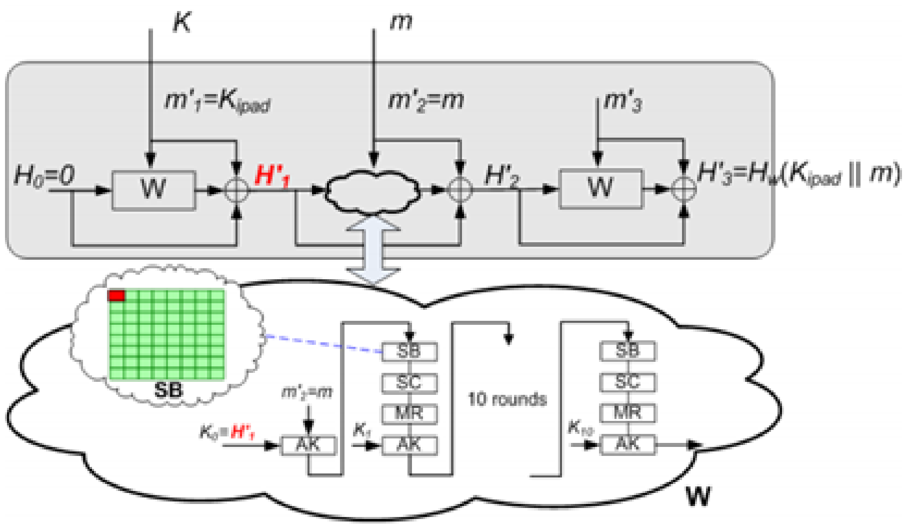


Figure 3.8: Target of selection function for power analysis attacks [13].

Side-channel Cryptanalysis of SHA-3 Candidates

Side-channel analysis was one of the main factors NIST took into consideration when selecting the winner of the SHA-3 competition. Benoit and Peyrin [19] study six second round SHA-3 candidates from a side-channel cryptanalysis perspective. The AES-based candidates include ECHO, Grøstl and SHAvite-3, and non AES-based candidates include BLAKE, CubeHash and HAMSI. The paper focuses on identifying an appropriate choice of selection function for each of the six candidates in an HMAC setting and evaluating the relative efficiency through simulations of correlation power analysis attacks on the different selection functions (AES S-Box, modular addition, HAMSI S-Box and exclusive XOR) based on the hash-inherent structure and internal primitives used. The selection functions mentioned in the previous list rank where an attacker should look for a power analysis attack. For example, to attack an AES-based SHA-3 candidate, one would look at the S-Box as the selection function due to its non-linear operation that corresponds to higher power consumption instead of the XOR operation that corresponds to lower power consumption. The authors draw conclusions concerning the relative complexity for protecting each candidate against first order attacks based on the elementary operations required to implement each hash function rather than show some particular hash functions can be broken with side-channel analysis.

By the third and final round of the SHA-3 competition, there were five remaining candidates. These were BLAKE, Grøstl, JH, Keccak and Skein. The work by Zohner *et al.* in [30] identifies side channel vulnerabilities for JH-MAC, Keccak-MAC and Skein-MAC and demonstrates attacks on the candidate reference implementations on an ATMega256-1 platform to evaluate the complexity and gain of the attacks. Additionally, the authors use profiling power analysis attack to recover the input to the Grøstl hash function, assuming the attacker does not possess knowledge of the input or output of the hash function and has only the measured power consumption data of a target cryptographic device. The paper does not conclude certain candidates are harder to attack in practice than others since the provided reference implementations are designed for understandability of the algorithm, not real-life practical implementation. As noted by the authors, the intention of the paper is to identify the operations an attack would exploit and to outline the necessity of side channel resistant hash functions in order to further encourage the development of countermeasures against side channel attacks on SHA-3 candidates [30].

Power Analysis Attacks of Keccak using Physical Traces

After NIST's announcement of Keccak as the winner of the SHA-3 competition, Taha and Schaumont published two papers where the first one is on side channel analysis of MAC-Keccak as discussed in [4] and the second paper looks at the application of differential power analysis of MAC-Keccak for any given key length as that can be referred to in [14]. As mentioned in the previous chapter, Keccak is built on sponge construction, and it provides a new Message Authentication Code (MAC) function called MAC-Keccak where the secret key is prepended to the message before the hashing. By obtaining the secret key through different means such as side channel analysis, it allows the attacker to forge the hashed message. In [4], Taha and Schaumont present a comprehensive side channel analysis of the MAC-Keccak where the secret key is used as part of the input message. The authors present analysis on the effect of changing the key length, all the possible attack points and complete attack scenarios. Taha and Schaumont show that the side-channel resistance of the MAC-Keccak depends on the key-length used, and they derive the optimum key-length as $((n \times rate) - 1)$, where $n \in [1 : \infty]$ and the $rate$ is the Keccak input block size. Besides discussing the side channel analysis theory of attacking MAC-Keccak, the paper applies the discussed theory in a practical attack experiment against MAC-Keccak implemented on a 32-bit Microblaze processor.

In their second paper [14], Taha and Schaumont look at how the changing of the key length impacts the set of internal operations that need to be targeted with differential power analysis. The proper selection of these target operations becomes a new challenge for MAC-Keccak when some key bytes are hidden under a hierarchical dependency structure. At the high level, they show the difference between the DPA of the target operations of MAC-Keccak and that of the typical cryptographic algorithms such as AES and HMAC that is illustrated in Figure 3.9. From the paper, the authors provide a complete differential power analysis of the MAC-Keccak for any key length by using a systematic approach to identify the required target operations [14]. Finally, Taha and Schaumont implement the Keccak reference software code on a 32-bit Microblaze processor built on top of a Xilinx Spartan-3E FPGA and successfully mount DPA attack by breaking several difficult case studies of MAC-Keccak with the results plotted in Figure 3.10. The authors chose the Keccak implementation with an input block size of $r = 1088$ and capacity of $c = 512$. The reasoning behind the choice of long key length is that it gives the MAC-Keccak implementation higher resistance to DPA. Complexity of attacking MAC-Keccak increases faster than linear with the key length due to the consecutive dependency between different key bytes. The usage of long key lengths are common in the cryptographic community such as in the applications of RSA. Overall, the two papers presented by Taha and Schaumont describe how to carry out attacks on a software implementation but not on hardware implementations.

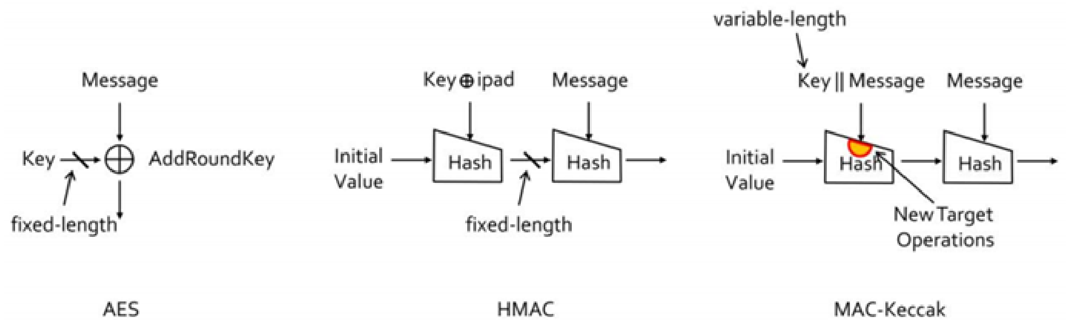


Figure 3.9: The difference between the CPA of AES, HMAC, and MAC-Keccak [14].

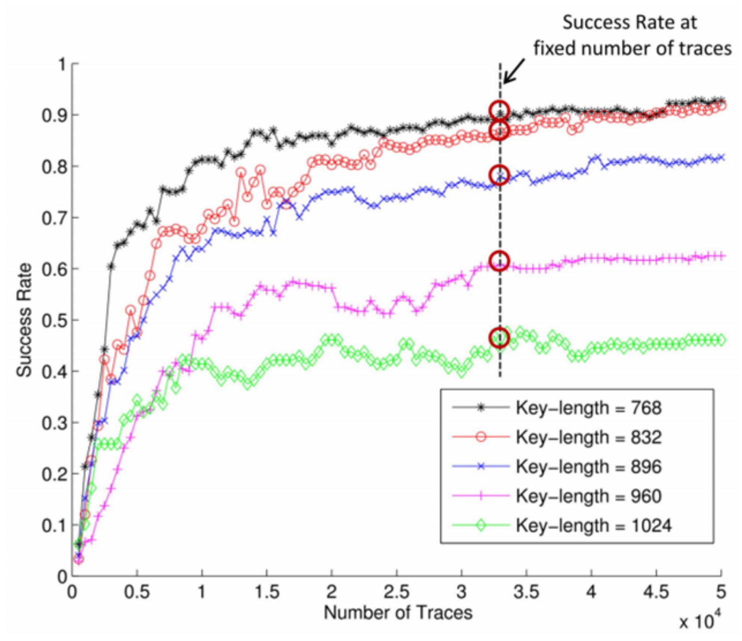


Figure 3.10: Success rate of different key length case studies [14].

Up to this point, there has been only existing work that focuses on attacks of software implementations of MAC-Keccak to retrieve the secret key, but there has been no comprehensive side channel vulnerability assessment of MAC-Keccak from the hardware implementation perspective. Pei *et al.* in [15] present the attack of targeting the θ transformation of the first round of MAC-Keccak implemented on an FPGA to retrieve the secret key that is 320 bits which is the size of the first plane, 40 bytes or 5 lanes. The authors construct several different side channel leakage models and implement CPA attacks based on them [15]. Recall the θ transformation in the previous chapter. The θ transformation is calculated in two phases. The first phase is for calculating the θ_{plane} which is the parity of each column and the second phase is for calculating the remaining part of the θ transformation such that it computes the XOR between every bit of the state and the two parity bits of the θ_{plane}. Of all the presented power models from [15], only power model 1 that targets the θ_{plane} and power model 3 that targets the output of the θ transformation provide high correlation values between the power model and the collected power traces. Pei *et al.* discuss the usage of power model 1 to recover all of the 320 key bits but it requires a significant number of power traces. In Figure 3.11, using power model 1 takes about 500,000 traces to recover one byte of one key lane with a success rate around 90%, with 8 bytes in each lane. To generate the power traces and apply power model 1 to produce the results in Figure 3.11, the authors implement the official VHDL code of unmasked Keccak implementation on a SASEBO board which contains a Xilinx Virtex-5 FPGA running at 12MHz and use an Agilent MSOX4104A oscilloscope to capture the measured power traces. The official Keccak VHDL code that is used is the high-speed core that has 1600 internal bits, with the rate $r = 1024$ bits and capacity $c = 576$ bits. The authors discuss the usage of power model 3 to recover the relationship, i.e. parity, between 2 key lanes, and there are 5 such relationships. This helps to decrease the key guess from 2^{320} to 2^{64}. Enumerating one lane with 2^{64} choices, one can find the the correct key from the parity of the two key lanes. The experiments from the paper show that power model 1 has lower correlation values than power model 3, plus it requires more traces in order to get high success rates for retrieving all of the 320 key bits correctly. Pei *et al.* recommend to use more than one side channel leakage models (two complementary models) to recover all the key bits of the MAC-Keccak such as using power model 1 to recover one key lane and then use power model 3 to recover the parity of the two adject key lanes of the target key lane that power model 1 successfully recovered. By knowing the key lane parity of the two adjacent key lanes of the target lane that produce the correct key from power model 1, the attacker could apply the reverse calculation of the θ transformation to retrieve the two adjacent key lanes that the attacker before only knows the key lane parity as a result from power model 3. The process repeats again until it retrieves the rest of the key lanes. The advantage of using two complementary models is that it increases the speed of recovering the secret key versus only using power model 1 in terms of using a fewer number of traces which means less CPA computations and shorter attack execution time.

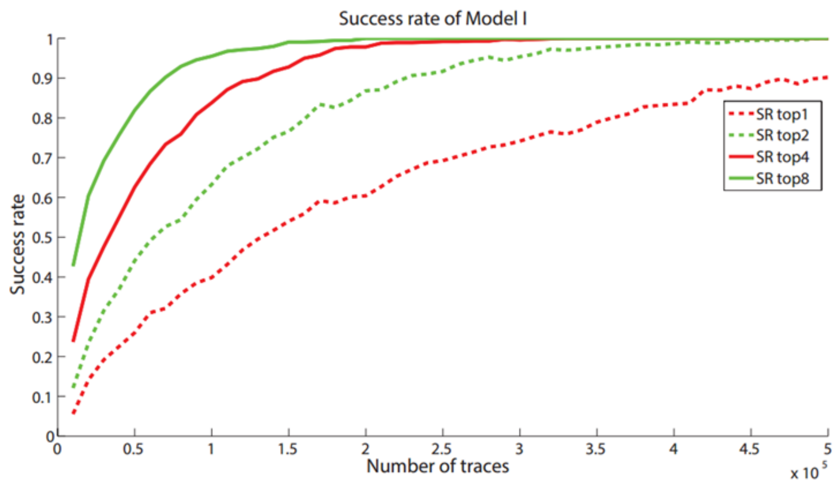


Figure 3.11: Success rate based on model 1 [15].

Chapter 4

Keccak Hardware Architecture

From the Keccak submission for the SHA-3 competition, Bertoni *et al.* provide three different flavors of Keccak hardware architecture implementions that target different platforms and these can be found in [2]. These three different hardware designs are: high-speed core, mid-range core and low-area coprocessor. The intention of this chapter is to focus on the high-speed core as the architecture looks very attractive with 1 clock cycle per round function and it is more practical to implement on real hardware devices. In addition, this is the main hardware architecture that this thesis focuses on attacking in the MAC-Keccak application to retrieve the secret key. It is still worthwhile to briefly discuss the other two Keccak hardware architectures, and the reader may consult [2] for more in depth discussion of these architectures.

Due to the symmetry and the simplicity of the Keccak round function, the algorithm design allows trading off area for speed and vice versa, such that different architectures reflect different trade-offs [2]. For each of the architectures, the authors provide two flavor instances of Keccak. The first one is the instance of Keccak with default parameter values and that is Keccak$[r = 1024, c = 576]$. It is built on top of Keccak-$f\,[1600]$ which is the largest instance of the Keccak-f family [2]. The second one is Keccak$[r = 40, c = 160]$, and it is the smallest instance of Keccak that makes use of Keccak-$f\,[200]$ such that its c capacity has sufficient level of security for many applications. The round function is based on simple boolean expressions and there is no need for adders or S-boxes with complex logic. Therefore, the benefit of avoiding the complex sub-blocks is to have a very short critical path for reaching very high frequencies [2]. As mentioned earlier, the design of the Keccak round function allows Keccak architectures to trade implementation area for speed.

From the name of the high-speed core, one can intuitively infer this core is the fastest among the three and requires a lot of silicon area in order to achieve that kind of performance. The in-depth discussion of the high-speed core will be looked at further in this chapter. The design purpose of the mid-range core is to be fast and also at the same time to save silicon area. Therefore, this core utilizes the folding technique by Jungk *et al.* in [31] on the Keccak round that results in reducing silicon area at the cost of performance. The implementation typically cuts Keccak's state into 2 or 4 pieces, naturally fitting between the fast core (1 piece) and the Jungk *et al.* compact implementation (8 pieces). As a result, the mid-range core circuit is not as fast as the high-speed core but more compact. The implementation is parametrized by N_b, which determines the amount of folding. With $N_b = 2$, the Keccak-f $[1600]$ permutation is computed in 74 clock cycles, and in 124 clock cycles with $N_b = 4$ [2]. At the other end of the spectrum among the three architectures is the low-area coprocessor that implements Keccak with the main focus of utilizing a small silicon area. Therefore, this architecture is ideal for wireless senor networks or smart cards where area plays an important role since it determines the cost of the device. Plus, on these small devices the operating system does run different processes in parallel and one does not expect high performance from these devices. In other words, the low-area coprocessor has a very small implementation area with low speed. In this architecture, the state of the Keccak is stored in memory and the coprocessor is equipped with registers for storing only temporary variables for calculations of a given round. Inside of the coprocessor, there are two parts: a finite state machine (FSM) and a data path. The data path is equipped with three registers for storing temporary values for round calculation, and the FSM computes the address to be read and sets the control signals of the data path [2]. From [2], it takes 215 clock cycles to compute one round of the Keccak-f permutation and out of these clock cycles, 55 cycles are used by the core for internal computation and do not transfer data in and out of memory.

Finally, the upper end of the spectrum among the three architectures in terms of performance is the high-speed core where one round of Keccak is computed per clock cycle. Due to this architecture characteristic, it is chosen as this thesis' main Keccak core, in particular utilizing the Keccak$[r = 1024, c = 576]$ instance with default parameter values to analyze and attack the MAC-Keccak configuration. The core operates in a stand-alone fashion such that the input block is transferred to the core, and the core does not use other resources of the system for performing the computation [2]. The advantage of this stand-alone hashing design is that the CPU can use direct memory access to transfer chunks of message to be hashed and while the core is computing the hash, the CPU can be assigned to a different task. The beauty of the high-speed core as shown in Figure 4.1, is that the computation of one Keccak-f round is based on plain combinational logic and is used iteratively for the different rounds. The three main noticeable components of the core from the figure are the input/output buffer, the state register and the round function. The use of the input/output buffer allows decoupling the core from a typical bus used in a system-on-chip (SoC) [2].

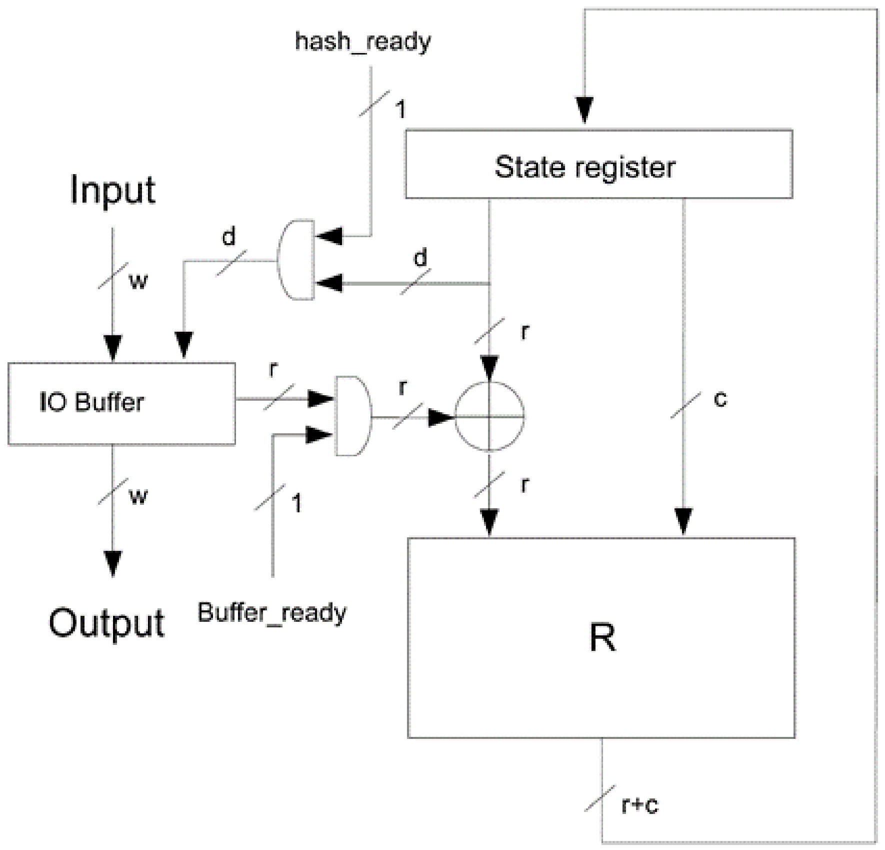


Figure 4.1: The high-speed core with one round per clock cycle [16].

The usage of the I/O buffer in the absorbing phase is to allow simultaneous transfer of the input through the bus and the computation of Keccak-f for the previous input block. Likewise in the squeezing phase, it allows the simultaneous transfer of the output through the bus and the computation of Keccak-f for the next output block [2]. The widths of the buses come in typically 8, 16, 32, 64 or 128 bits. The provided high-speed core by Bertoni *et al.* for the SHA-3 competition uses the Keccak$[r = 1024, c = 576]$ instance where width of the bus is fixed to the lane size w of the underlying Keccak-f permutation. Thus, this limits the throughput of the sponge engine to w per cycle [2].

The structure of the standard Keccak hardware implementation in a high level block diagram is shown in Figure 4.2, where one state register holds both the input and output of a round operation that consists of five transformations. The names of the components in this figure reflect the actual components inside of the VHDL reference source code provided by Bertoni *et al.* A test bench is written to feed in the message blocks from a text file, keccak_in.txt, into the Keccak core, where the core performs the hash computation. The test bench is also responsible for writing the hashed output messages to the output file, keccak_out.txt. These functions and tasks of the test bench are captured in a simple state machine diagram that is depicted in Figure 4.3. A small example of an input file with seven messages to be hashed can be seen in Figure 4.4. The format of the file is fairly simple such that the first line dictates the number messages in the file, each row in hex values represents a single row of 8 bytes, a single '' denotes an end of the current block message and, finally, a sequence of '-' and '.' notify the test bench that it is the end of the input file. Since the Keccak instance that is used in this thesis has the rate $r = 1024$, there are 16 rows of 8 bytes each, to reflect the rate size. The sequence of the rows indicates the order they are used to fill the Keccak state, i.e. one lane at a time, starting from the bottom plane and then going upward. The message block that is expressed in 16 rows is padded according to Algorithm 9 where the last bit of the rate has to be '1' and there may be 0 or more '0' bits in between the last bit '1' of the rate and the preceding bit 1. Each row is 8 bytes or 64 bits which is the size of w for Keccak-f $[1600]$. The corresponding hashed output messages for the simple message input in Figure 4.4 is captured in Figure 4.5. The format of the file lists the four rows as the output message and the '-' indicates the end of that current hashed output message. Similar to the input file, each row of the output file is expressed in hex and is 8 bytes long. The output is expressed in four rows and is 256 bits long.

Thus far the discussion is at the high level, using the test bench to feed in the inputs and then save the corresponding hashed output message into a file. As mentioned before, Figure 4.2 is the high level block diagram of the Keccak high-speed core hardware implementation. At the beginning of the hash operation is *keccak_buffer* which is the IO buffer that reads in the message from the keccak_in.txt. Since a single message block in the input file is expressed in 16 rows, it takes *keccak_buffer* 16 clock cycles to absorb the message. Upon the next clock cycle, the *din_buffer_full* signal of the *keccak_buffer* goes high and that is an indication to XOR the 1024 bits that were read in with the state register that is initialized to 0s as inputs to the combinational logic five-step operation as this is the first round of Keccak. The output of the five-step operation is written back to the state register. The following 23 rounds of Keccak take the state register output as input to the combinational five-step operations. At the end of the last round of Keccak, when the $dout\_valid$ signal of the $keccak$ component is high, it is an indication to the test bench to start recording the output message, where the output message is squeezed in four clock cycles and that is also when the $dout\_valid$ signal goes low. Note from Figure 4.2, the $dout\_valid$ signal of the $keccak$ component is nothing more than the *dout_buffer_out_valid* output signal from the *keccak_buffer* component, saying the output data is ready. Each clock cycle that the core squeezes an output, it is 64 bits. Once the test bench completes writing the hashed output message to file, it resets the core and waits a few clock cycles before repeating the process of hashing the next message block from the file. The total hashing requires 47 clock cycles, as summarized in Equation 4.1, where it takes 16 clock cycles to absorb the message, 24 clock cycles to iterate through 24 rounds of five-step transformations, 3 clock cycles to wait for the I/O buffer to indicate the output is valid to read, i.e. sets $dout\_valid$ active, and finally 4 clock cycles to squeeze the output.

$$\begin{aligned} Total\_Cycles &= Read\_Message + [Rounds\ \times\ (Number\_Cycles\_Per\_Round)] + \\ &\qquad Wait\_Dout\_Valid\_Active + Write\_Hashed\_Output \\ &= 16 + [24\ \times\ (1)] + 3 + 4 \\ &= 47 \end{aligned} \tag{4.1}$$

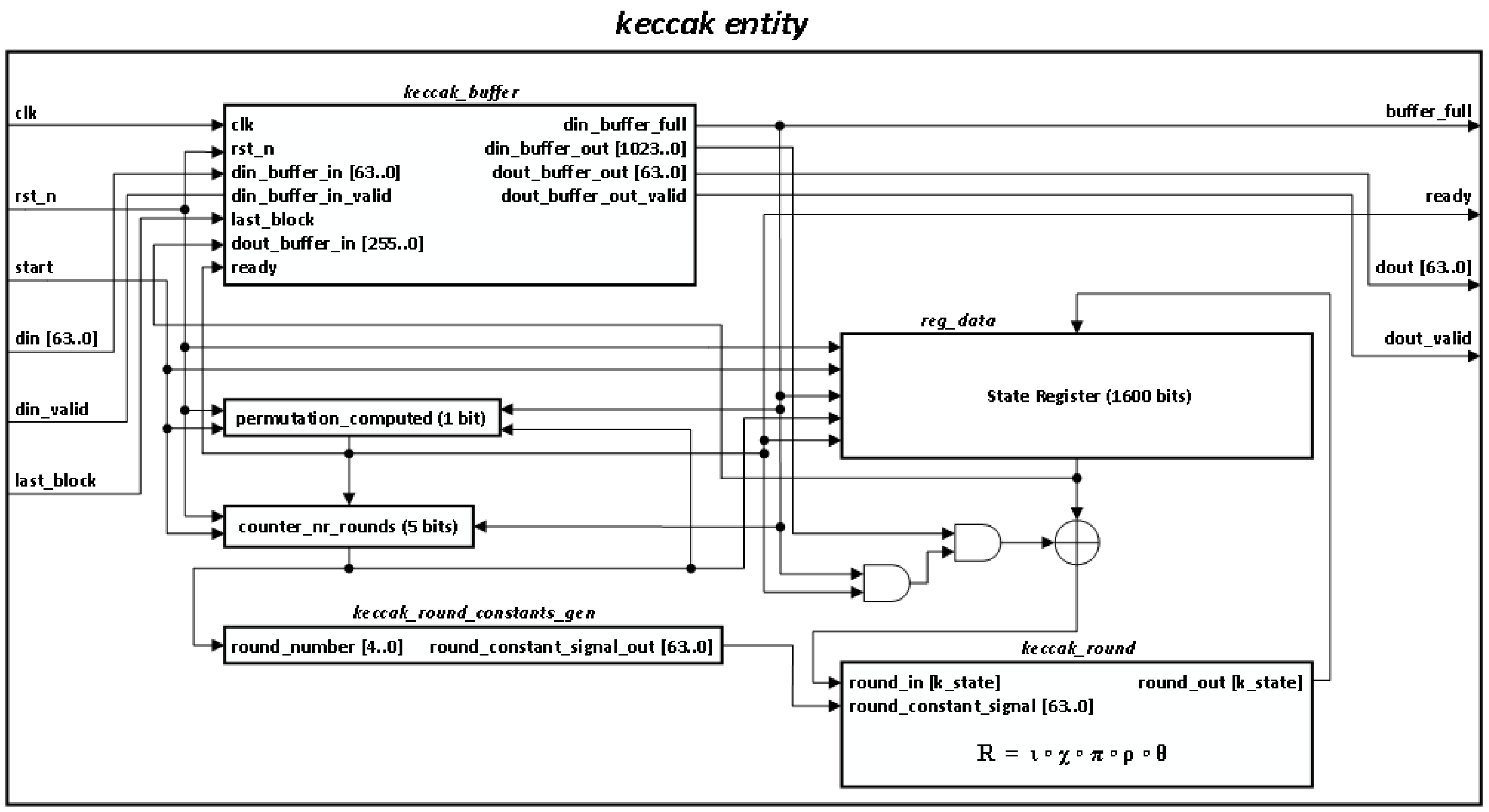


Figure 4.2: The high-speed core with one round per clock cycle [16].

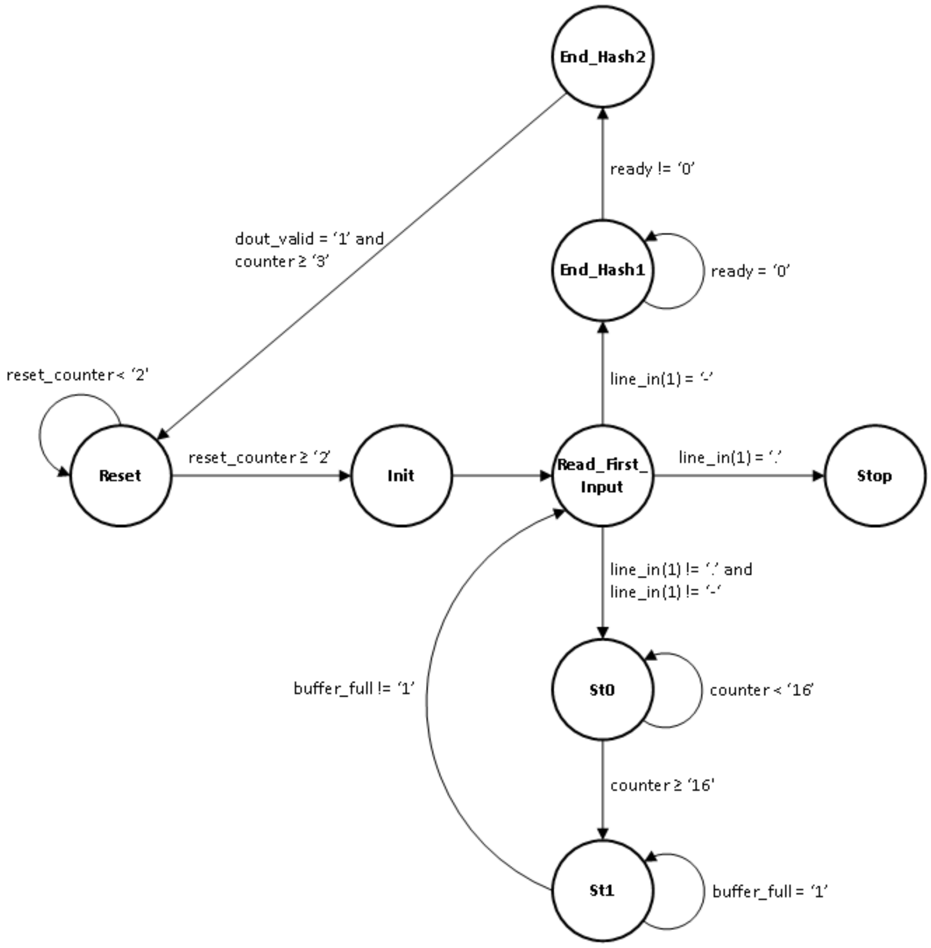


Figure 4.3: The high-speed core test bench state machine [16].

```
7
0706050403020100
0F0E0D0C0B0A0908
1716151413121110
1F1E1D1C1B1A1918
2726252423222120
BFE7AE43B614180D
2B675BB3D8C601DD
8159BEF30620CE59
3DC1001F6242A04A
EE43C1D7E0AFC9BF
B5A991C1C9CE538A
CC6177053B6C79C2
FDFD6ECFA7392C42
26B1E3D152479029
BCD838BED58E7C59
8000000000000001
–
        . . .
        . . .
        . . .
–
0706050403020100
0F0E0D0C0B0A0908
1716151413121110
1F1E1D1C1B1A1918
2726252423222120
36586E33363C7A0B
C0B0FA131183077A
4241EEFD30D5A3A5
899DF0E316307700
5857F99DFC57C49E
E489D7684E515BB9
574FAA622DE82530
514A670FEC2B9C04
621E57D793ED4FDB
92F499855E43627D
8000000000000001
–
.
```

Figure 4.4: Input file, keccak_in.txt, used by the test bench to feed the messages in the file into the Keccak high-speed core for hashing. A single message block of 1024 bits in the file consists of 16 rows where each row, represented in hex, is 8 bytes. A row corresponds to a single lane in the Keccak state.

```
EAFC4EF0141306BC
3F6A1946CCB7E1AC
7536B073314274E2
8C318CCC8CEA4786
-
        . . .
        . . .
        . . .
-
E8DDF1FC40380B8C
6E3A5A0A240377CC
AC183172B55B76FF
2FD13EDE76F54227
-
```

Figure 4.5: Output file, keccak_out.txt, created by the test bench to store the hashed output messages corresponding to the input messages in keccak_in.txt. The length of each hashed output message is 256 bits, represented by 4 rows in the output file where each row, expressed in hex, is 8 bytes.

Just to recapture the discussion of how many clock cycles it takes to hash a block of message that happens to be the size of the rate r, the following figures, Figure 4.6 and Figure 4.7, show simulation waveforms looking at the hashing operations that use 24 rounds and another waveform to focus on the first few rounds. The simulation waveforms are generated by the ModelSim tool, and cursors are set at points of interest for analysis. The example message block that is used in these two ModelSim waveforms and also its corresponding output hash value came from the input and output files depicted in Figure 4.4 and Figure 4.5, respectively.

Figure 4.6 shows the capturing of the 24 rounds of Keccak for hashing a message. At time 80ns, the Keccak high-speed core starts to read in the message at one lane, i.e. 64 bits, per clock cycle, where a single clock cycle is 20ns in this instance. By 400ns, the message is already read in and the core begins to start the hash computation with round 1. The cursors at 420ns and 440ns correspond to the start of the second and third round, respectively. The time at 1020ns denotes the end of squeezing the output to the file. Finally, the test bench resets the core and lets it settle down a few cycles before starting the whole hash process for the next message in the input file that begins at 1120ns. Note that at time 940ns, the $dout\_valid$ of the $keccak$ component goes high indicating output data is ready, and that is when the core squeezes the output and also when the test bench saves the output to the file. The reason for the test bench resetting the core at the end of the hash operation and also consuming a few clock cycles after the reset is to make sure everything is resetting back to normal and, specifically, to make absolutely certain the state register only contains 0s. It is critical for the state register to be all 0s at the beginning of each hash computation because the attacked power model relies on the change of state values in order to attack Keccak successfully. This will be elaborated more in the next chapter regarding the simulation and attack frameworks. Figure 4.7 is nothing more than a zoomed-in of the curves of the previous figure, looking specifically at the first few rounds. The purpose of this is to understand the exact timing of when Keccak completes two round versus twenty-four round hash computation in order to compare the application, i.e. CPA, attack time between using power traces from these two different rounds.

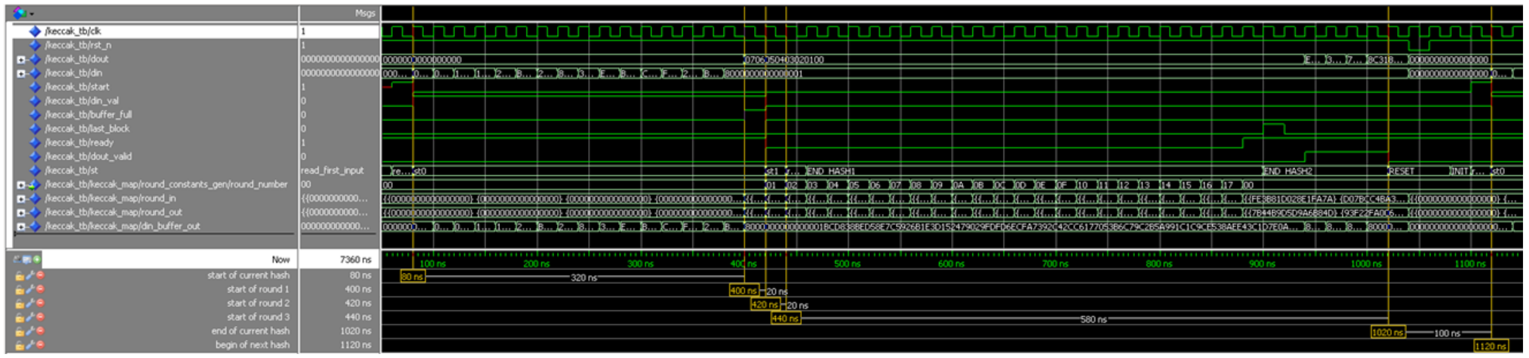


Figure 4.6: Simulation waveforms for hashing a block message of 1024 bits looking at all 24 rounds of Keccak.

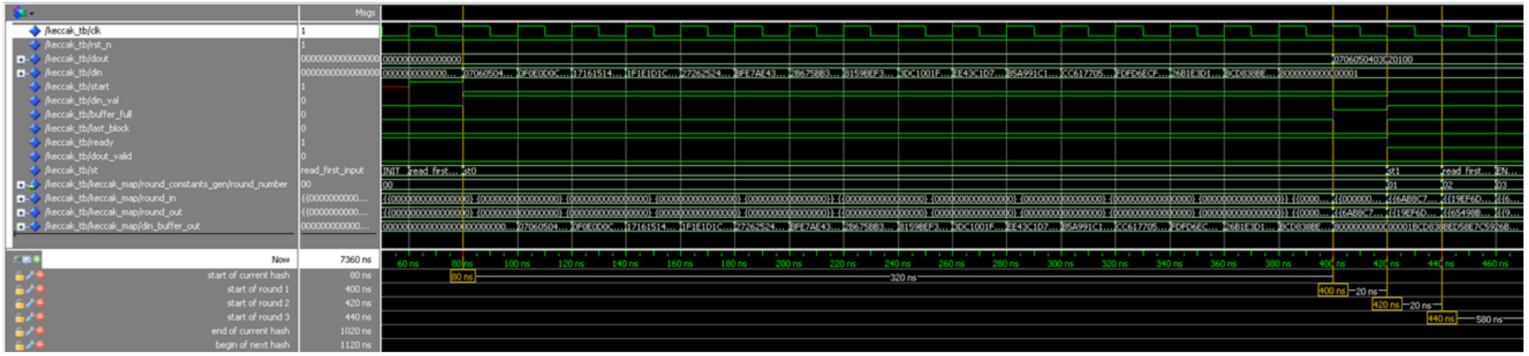


Figure 4.7: Simulation waveforms for hashing a block message of 1024 bits looking at first 2 rounds of Keccak.

Chapter 5

Simulation and Attack Frameworks

5.1 Simulation Framework

Compared to the software implementations of MAC-Keccak implemented in [4] and [14], hardware implementations of MAC-Keccak with their parallel implementations have much lower side channel leakage. Therefore, side channel attacks on a hardware implementation of MAC-Keccak are much more challenging to implement as can be seen in [15] by Pei *et al.* So far, there are attacks of physical hardware implementations of MAC-Keccak using FPGA development boards, but there has been no side channel vulnerability assessment of the hardware implementations using simulated power consumption waveforms. When compared to physical power extraction, circuit simulation significantly reduces the complexity of mounting a power attack, provides quicker feedback during the implementation/study of a cryptographic devices, and that ultimately reduces the cost of testing and experimentation. Therefore, power analysis with simulated traces enables the evaluation of a protected and unprotected cryptosystem's effectiveness against power attacks at various stages of the design with significantly less overhead compared to the study of a physical device [7]. On the other hand, there are some concerns that need to be kept in mind when generating and using simulated power traces for power analysis attacks, those are maintaining power trace sample precision, i.e. ns versus ps time domain, or 5 decimal place power values versus 2 decimal place power values, and producing compact output power trace files as quickly as possible. This research simulates the execution of the Keccak high-speed core provided by Bertoni *et al.* for the Keccak submission to the SHA-3 competition, where the simulation framework to produce the power traces of the target ASIC implementation builds off the design and simulation flow established in the work by Ken Smith in

[9] and Garrett Smith in [7]. The research uses the fundamental simulated power trace design flow from Ken Smith and uses the Makefile script concept from Garrett Smith for coordinating different tools in Ken Smith's power trace design flow for efficiency and code maintainability.

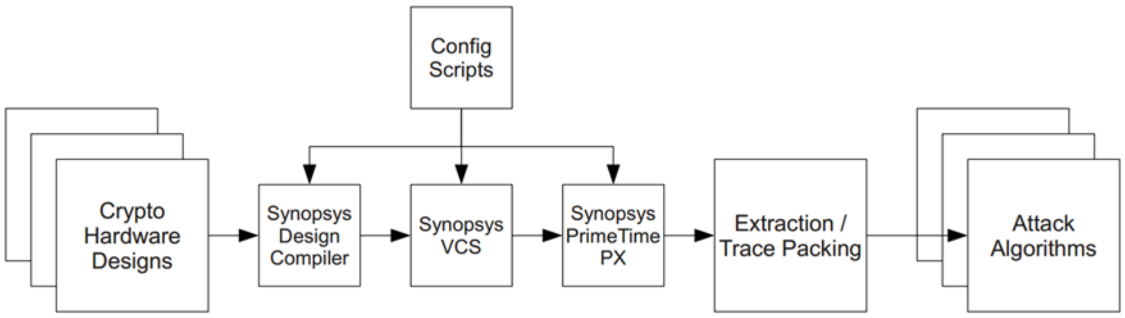


Figure 5.1: ASIC top-level simulation flow [9].

Figure 5.1 illustrates the top-level simulation flow used by Ken Smith in [9], and it is also the same simulation flow that is leveraged by this thesis to generate power traces for the Keccak high-speed core. The design flow utilizes the Synopsys electronic design automation (EDA) tools to compile, synthesize, simulate and perform power estimation of the hardware design. The simulated power extraction process is split into three main steps. First, the hardware design is compiled and synthesized using the Synopsys Design Compiler. After the design is built, the synthesized implementation and its corresponding test bench are then simulated using Synopsys VCS. During the simulation, the algorithm is executed many times with different input vectors. Finally, the simulation output is processed by Synopsys Primetime PX tool to generate a time-based power report [7]. Various configuration files are used to specify the design sources to build, the technology library and the various parameters relevant to design simulation. The input vector, the extracted waveform from the simulation and the output data, i.e. ciphertext from block cipher encryption or digest output from hash algorithm, are then grouped and stored until they are required for evaluation. Several other utilities are used for supporting the simulated power extraction, and all of these tools are implemented to run on the CentOS Linux workstation. The above described the process to generate simulated power traces and store/compress the traces along with their corresponding messages. Hashed messages are controlled by the global shell script which is shown in Figure 5.2. The shell script is invoking the main build target, $power$, from the Makefile script as shown in Figure 5.3 to generate the simulated power traces as a result of invoking other dependent targets that are on the list of Makefile-chained build targets to compile, synthesize, simulate and perform power estimation of the Keccak high-speed core hardware design. The original simulation in [9] by Ken Smith uses Perl scripts to achieve the objective of generating simulated traces, but these neither have the flexibility of invoking intermediate build targets nor the dependency of tracking intermediate files that are used by the final build target, i.e. files used by PrimeTime PX to generate traces. Therefore, this research leverages the idea of using the Makefile script to enable dependency tracking of the intermediate resources from [7] by Garrett Smith. Not only is using Makefiles useful for tracking the dependency of intermediate resources, it also allows the flexibility to invoke the intermediate sub-targets that build these resources, which is very useful when debugging compilation, synthesis or simulation issues.

```
#!/bin/bash

# Defined output folder.
folder=Power_Traces_$(date +"%m_%d_%Y.%Hh_%Mmin")

# Print start banner.
echo "**** Power Attack Simulations ****"

# Create output folder.
mkdir $folder
sleep 5

# Iterate through the list of keccak_in_*.txt and generate power traces.
for iter in $(seq -f %03g 1 10)
do
    # Remove any unnecessary old files before generating any
    # power traces.
    make clean

    cp test_vectors/keccak_in_$iter.txt keccak_in.txt

    # Call 'power' target from the Makefile to compile, simulate, and
    # generate power traces.
    make power

    # Strip power_waveform.out for unnecessary information i.e. banner.
    ./parser_out_file.pl power_waveform.out $folder/power_waveform_$iter.out

    # Move artifacts to output folder for archiving.
    mv simulation.txt $folder/simulation_$iter.txt
    mv keccak_out.txt $folder/keccak_out_$iter.txt

    # Let the user know the power trace generation for current
    # keccak_in.txt is done.
    echo "**** Completed power simulation for input file - keccak_in_$iter.txt
     ****"
    sleep 10
done

# Remove any unnecessary old files.
make clean

# Move artifacts to output folder for archiving.
cp -r test_vectors/* $folder

# Perform quick checksum on the artifacts in the output folder and store
# those checksums for future reference if need to check the file
# integrity of any artifact file.
sha256sum $folder/* > $folder/cksum_sha256sum.txt

# Zip up the output folder to save memory space.
tar -zcvf $folder.tar.gz $folder
```

Figure 5.2: Global script for coordinating HDL compilation, generating power simulation traces, and zipping artifacts for software attack.

```
###########################################################################
# Defined variables.
###########################################################################
QUIET ?= 0
CLOG = compile.log
SLOG = simulate.log

ifeq ($(QUIET), 1)
    CLOG += &>/dev/null
    SLOG += &>/dev/null
endif

DESIGN_HDL = lib/core.v keccak_netlist.vhdg tb/keccak_tb.vhd

###########################################################################
# Rule to create the gate-level netlist.
###########################################################################
keccak_netlist.vhdg: scr/dc_commands_keccak.tcl lib/core_typ.db
    dc_shell -f $< | tee $(CLOG)

###########################################################################
# Rules for compilation, simulation, and power analysis.
###########################################################################
simv: $(DESIGN_HDL)
    vlogan lib/core.v | tee -a $(CLOG)
    vhdlan keccak_netlist.vhdg | tee -a $(CLOG)
    vhdlan tb/keccak_tb.vhd | tee -a $(CLOG)
    vcs -debug -verb keccak_tb | tee -a $(CLOG)

dump.vpd: scr/simv_commands_keccak.tcl simv
    ./simv -ucli -do $< +ntb_random_seed_automatic | tee $(SLOG)

dump.vcd: dump.vpd
    vpd2vcd +morevhdl +includemda $< $@ | tee -a $(SLOG)

power_waveform.fsdb: scr/pt_commands_keccak.tcl dump.vcd
    PT_PIPE=0 pt_shell -f $< | tee -a $(SLOG)

power_waveform.out: power_waveform.fsdb
    fsdb2ns -fmt out -o $@ $<

.PHONY: power
power: power_waveform.out

###########################################################################
# Rule to clean up and remove unnecessary files.
###########################################################################
.PHONY: clean
clean:
    rm -rf AN.DB 64 csrc work simv simv.daidir transcript* *.txt *.out *.vhdg \
        *.ini *.key *.wlf *.log *.vcd *.vpd *.svf *.fsdb fsdb2nsLog \
        ARCH ENTI *.mr *.syn PACK
```

Figure 5.3: Makefile for simulated power trace generation.

5.1.1 Synthesis

Hardware compilation and synthesis of the Keccak high-speed core require the use of several tools, illustrated in Figure 5.4. Tools are represented as circles and resources such as scripts, sources and intermediate files are expressed as rectangles. The build target of the design synthesis is the compiled simulation executable, $simv$, which is invoked by the Makefile in Figure 5.3. Basically, when the Makefile calls $simv$, it is simply invoking the various tools to generate the $simv$ binary. The first dependent target to generate $simv$ is $keccak\_netlist.vhdg$. The Makefile rule for this target invokes Synopsys dc_shell tool. The Keccak high-speed core source files are read in and parsed by the dc_shell tool. The tool then synthesizes an implementation netlist built from standard cells provided by the Synopsys $core\_typ.db$ 130nm library. This netlist is then written back out into $keccak\_netlist.vhdg$. The $vhdlan$ tool then parses the generated netlist and the top-level test bench. The standard cell library Verilog model $core.v$ is parsed with the $vlogan$ utility. A simulation executable $simv$ is created with the VCS tool. The above discussed steps can be seen with the rule for the $simv$ target in the Makefile which shows the list of commands that need to be executed for hardware synthesis. Another thing to note from the $simv$ build target rule is the need to specify the hardware test bench at the top-level for the simulation. This is done so the test bench can operate the unit under test which is the Keccak high-speed core hardware implementation. Figure 5.5 shows the commands executed by dc_shell to synthesize the hardware design from the standard cells. Note that the hardware test bench is excluded from the synthesis.

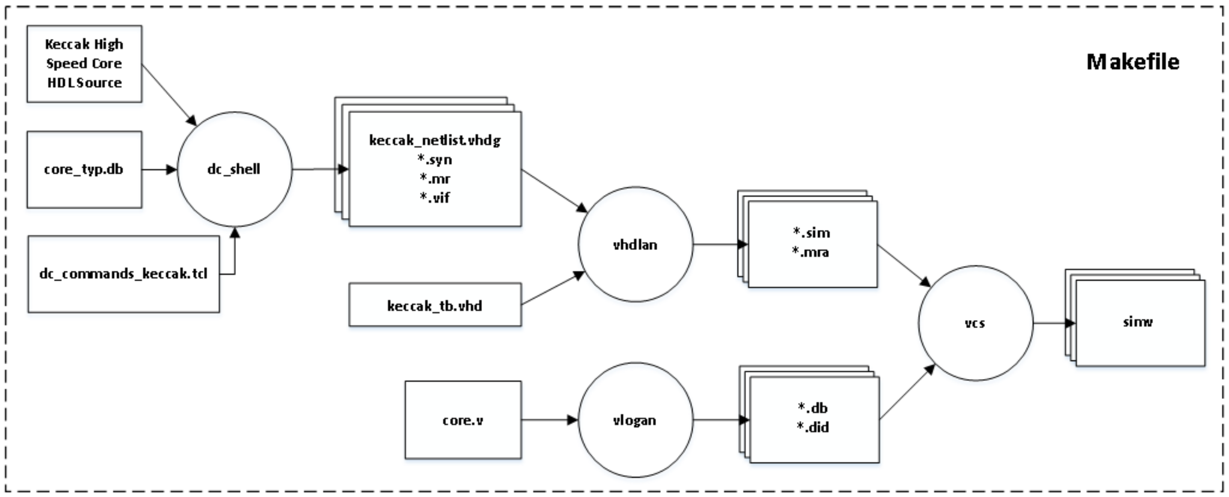


Figure 5.4: Hardware synthesis and simulation executable generation flow.

```
###########################################################################
# Link design.
###########################################################################
set search_path {rtl tb lib .}
set target_library {core_typ.db}
set link_library {* core_typ.db}

###########################################################################
# Compile HDL sources.
###########################################################################
set compile_clock_gating_through_hierarchy true

read_file -format vhdl "keccak_globals.vhd"
read_file -format vhdl "keccak_round_constants_gen.vhd"
read_file -format vhdl "keccak_round.vhd"
read_file -format vhdl "keccak_buffer.vhd"
read_file -format vhdl "keccak.vhd"

current_design keccak
link

list_designs
uniquify -force

check_design
compile -map_effort medium

report_cell
report_area -hierarchy

###########################################################################
# Generate the gate-level netlist.
###########################################################################
write -format vhdl -hierarchy -output keccak.vhdg

exit
```

Figure 5.5: Hardware synthesis script for the Keccak design.

5.1.2 Power Simulation

There are three steps to simulate a design using the Synopsys VCS MX-based flow and these involve analysis, elaboration and simulation. The analysis and elaboration steps described in the previous section build the Makefile target $simv$, also known as the simulation executable. The simulation executable gets executed many times with varying stimuli from an external file, $keccak\_in.txt$. The execution of the $simv$ is driven from the Makefile which automates the process of simulating the design and invoking PrimeTime to generate the power waveforms and place them in the $power\_waveform.out$. Figure 5.6 is a diagram of the simulation flow for power trace generation. The $attack.sh$ script is the main script for invoking the main build target, $power\_waveform.out$, in the Makefile and, once the main build target is built, it is responsible for stripping the raw power trace file of unnecessary information and then compressing/zipping the trace files for later usage in the CPA attack application, then invoking the whole process again if there is any more data in the $keccak\_in.txt$ file.

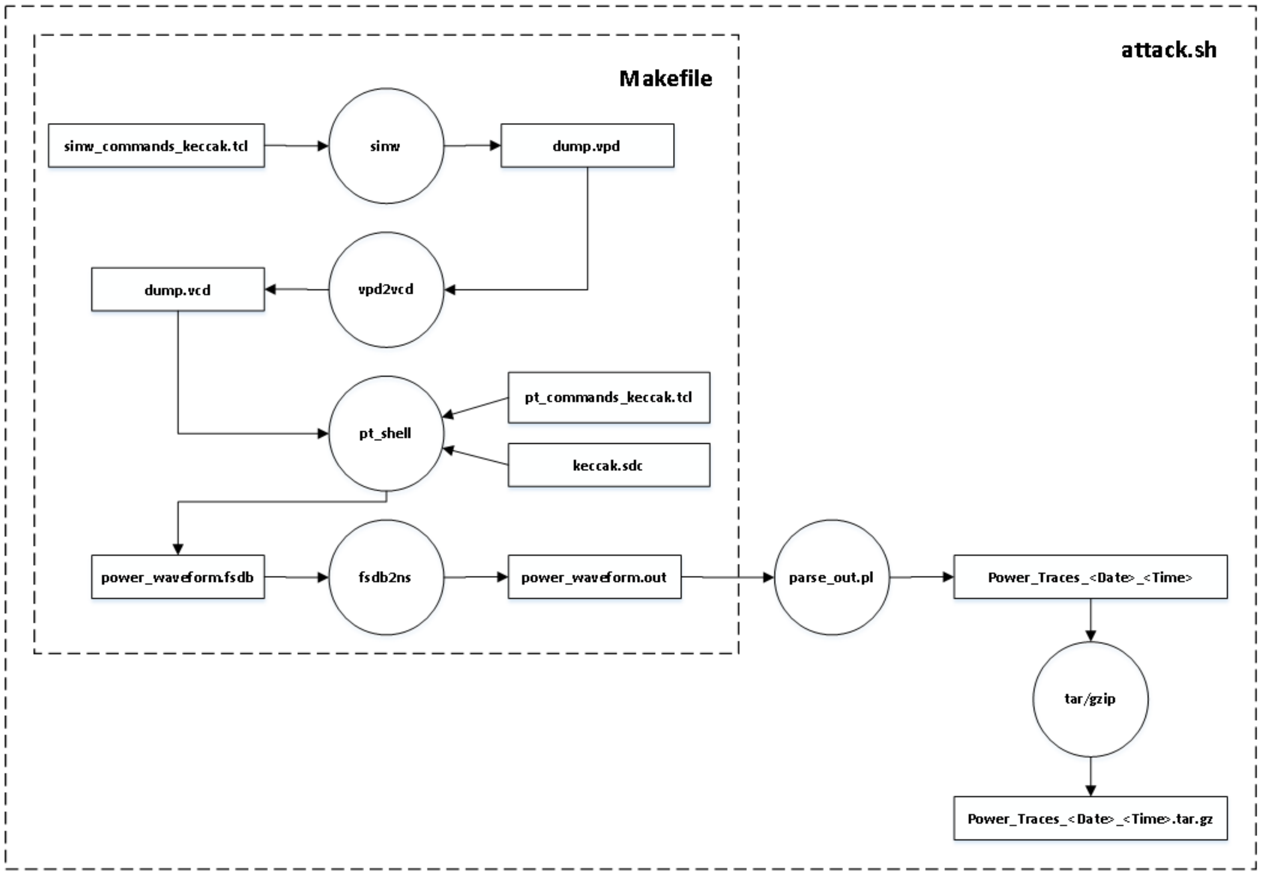


Figure 5.6: Simulation and power modeling flow.

$simv$ takes a $simv\_commands\_keccak.tcl$ file as input, which is used to specify the simulation runtime, time resolution and value change dump file as can be seen in Figure 5.7. It is critical for the simulation executable to dump the signal values after every step in time to $dump.vpd$. The test bench is responsible for loading input test vectors and writing the result output to a file. The input vectors were generated externally using the provided C code program by the Keccak developers, Bertoni *et al.* The program is modified to write an arbitrary number of test vectors specified by the user into the $keccak\_in.txt$ file where each message is padded according to the padding scheme. This file is parsed by the VHDL test bench to provide stimuli for the Keccak high-speed core to hash. The length of the simulation runs for a maximum of 1,000s at the default resolution of 10ps. Whenever the test bench reaches the end of $keccak\_in.txt$ with no more input files, it will assert and stop the simulation and not run for the remainder of the 1,000s. The only reason for specifying the large amount of time is to be able to simulate some test cases where the *keccak_in.txt* might have 50,000 messages. In this research, each *keccak_in.txt* file is capped with 20,000 messages to be hashed for power traces.

```
dump -file dump.vpd
dump -add /KECCAK_TB -depth 0
dump -autoflush on
dump -deltaCycle on
dump -forceEvent on
run 1000000000000
exit
```

Figure 5.7: Simulation executable commands.

With each simulation, the *simv* executable produces a VPD (Value change Plus Dump) file. This is a Synopsys proprietary binary format that captures the changes in value of signals within a design over the course of a simulation. Plus, it is a more compact form compared to the ASCII-based IEEE standard VCD (Value Change Dump) file format. For the VPD to be usable by PrimeTime, it first needs to be converted to VCD format by using the *vpd2vcd* tool. Under the hood, the PrimeTime analysis engine operates on VCD activity data. The usage of the Synopsys PrimeTime PX is invoked by using the *pt_shell* environment. The *pt_shell* tool processes a Tcl script, as in Figure 5.8, that configures the environment for time-based power analysis. In the time-based mode, PrimeTime examines how the signal values change over the course of the simulation, gathered from the value change dump, and computes the instantaneous power consumption for each simulation event. Another part of the power analysis for PrimeTime is to provide an SDC constraint file with the system clock frequency which in this research is specified with a 20ns period.

PrimeTime first parses the standard cell library, design netlist and design constraints file. The activity information is then loaded using the *read_vcd* command, which also allows the tool to determine the mapping between the VCD signals and the objects in the design netlist. To perform power analysis calculations of each entity in the Keccak high-speed core design, only, without mixing with the test bench signals in PrimeTime, the VCD is read using the *read_vcd* command with the *strip_path* parameter specified. Power analysis is performed by calling the *update_power* and *report_power* commands. Prior to analysis, $set\_power\_analysis\_options$ is used to specify which objects from the design hierarchy should be monitored for waveform generation and also the format of the waveform file containing the instantaneous power consumption data. The top level design is sufficient in this research to monitor and report. The results are written to a FSDB waveform file which is a proprietary binary file format. The *fsdb2ns* tool is used to convert the FSDB file into an ASCII format OUT file that will be useful in the attack phase for the evaluation algorithms to extract and access the waveform information. The *fsdb2ns* tool can convert from FSDB to OUT format without a loss in precision [9]. One important thing to note about the PrimeTime tool is that it reports power consumption for each simulation event when a signal changes value as opposed to using a fixed sampling rate. Each power event is recorded in the .out file as a time index and its corresponding power sample. A *parse_out.pl* Perl script was written to process the waveform produced after each simulation such as stripping out unnecessary information like duplicate channels of information and some header fields. This new processed waveform is then compressed and zipped along with the $keccak\_in.txt$, $keccak\_out.txt$, and $simulation.txt$ files in order to save disk space, as generating and storing large volumes of trace data requires a significant amount of storage [7]. The purpose of grouping these files and then zipping them is a simple way to associate these pieces of information for the evaluation algorithms to use. Each simulation is independent, so the process may be parallelized by invoking multiple instances of $attak.sh$, simultaneously. This will reduce the simulation time by a factor of N where N is the degree of parallelism.

For each simulation, three output files are produced: a Nanosim .out formatted waveform containing the combined power traces for each of the hash operations simulated in $power\_waveform.out$, a timestamp archive containing a single record for each individual hashed message operation in $simulation.txt$, and the hashed outputs in $keccak\_out.txt$. Each record contains the hash operation's start time and input message as shown in Figure 5.9. The CPA attack framework requires these timestamps in order to locate and extract each power waveform correctly. The starting timestamp is subtracted from each event timestamp in order to properly align the trace samples for evaluation.

```
##########################################################################
# Set the power analysis mode.
##########################################################################
set power_enable_analysis             true
set sh_source_uses_search_path        true
set power_read_activity_ignore_case true
set power_analysis_mode               time_based

##########################################################################
# Set library search path, read in the netlist, and link the design.
##########################################################################
set search_path     "lib ."
set link_library    "* core_typ.db"
read_vhdl           keccak.vhdg
current_design      keccak
link

##########################################################################
# Run timing analysis, read in constraints and switching activity file.
##########################################################################
update_timing
read_sdc lib/keccak.sdc

set waveform_path "power_waveform_old"
read_vcd -strip_path KECCAK_TB/KECCAK_MAP dump.vcd

##########################################################################
# Perform power analysis.
##########################################################################
set_power_analysis_options -waveform_format fsdb             \
                           -waveform_output $waveform_path   \
                           -waveform_interval .01            \
                           -include top

check_power
update_power
report_power
report_power -hierarchy

quit
```

Figure 5.8: PrimeTime PX simulation script for the Keccak design.

```
80  ns
          0706050403020100
          0F0E0D0C0B0A0908
          1716151413121110
          1F1E1D1C1B1A1918
          2726252423222120
          BFE7AE43B614180D
          2B675BB3D8C601DD
          8159BEF30620CE59
          3DC1001F6242A04A
          EE43C1D7E0AFC9BF
          B5A991C1C9CE538A
          CC6177053B6C79C2
          FDFD6ECFA7392C42
          26B1E3D152479029
          BCD838BED58E7C59
          8000000000000001

          ...
          ...
          ...

6320  ns
          0706050403020100
          0F0E0D0C0B0A0908
          1716151413121110
          1F1E1D1C1B1A1918
          2726252423222120
          36586E33363C7A0B
          C0B0FA131183077A
          4241EEFD30D5A3A5
          899DF0E316307700
          5857F99DFC57C49E
          E489D7684E515BB9
          574FAA622DE82530
          514A670FEC2B9C04
          621E57D793ED4FDB
          92F499855E43627D
          8000000000000001
```

Figure 5.9: Simulation timestamps with input messages for Keccak high-speed core with 24 rounds.

5.2 Attack Framework

There was a great deal of value in performing initial testing and verification of the attack framework's analysis algorithms by reproducing existing successful power analysis attacks such as reproducing the simulated attacks on the AES block cipher from the work of Ken Smith in [9] and Garrett Smith in [7]. The top-level class diagram of the attack framework is illustrated in Figure 5.10. The attack framework was designed in a modular, plugin-oriented fashion in order to facilitate the addition of support for new cryptosystems, leakage models and attack algorithms. The framework is written in C# using Microsoft Visual Studio to take advantage of the powerful debugging tool, IntelliTrace, with capabilities such as interactive code tracing/stepping. The other benefit of using C# in Microsoft Visual Studio is for code maintenance and flexibility that future work can expand on. As of right now, the framework is able to support power analysis attack on AES and Keccak cryptosystems with simulated and physical power traces. Each of the attack algorithms, such as DPA and CPA, is written to have a built-in leakage models such as Hamming weight and trace parsers, this way if there is a change to one trace parser such as for reading simulated traces for Keccak, CPA will not impact the other trace parser for reading the physical traces of Keccak CPA. The current framework supports the attack of AES with DPA 1-bit attack and CPA 8-bit attack. Likewise, for Keccak, the current framework supports CPA 1-bit, 2-bit, 4-bit, 8-bit and 16-bit attacks. The framework also has the graph and utility instances that are used to plot results from the SPA, DPA or CPA attack algorithms such as SPA power curve, DPA confidence ratio and CPA success rates, just to name a few. The central attack manager is the SCA (Side Channel Analysis) module that is responsible for the creation and distribution of work. Based on the number physical CPU cores available on a given workstation, the SCA module will spawn one or more worker threads, up to the number available CPU cores, for the attack. The usage of multiple threads in CPA makes a big difference between completing an attack in a few hours with multiple threads and a few days for a single thread due to the intensive computation for correlation between the power traces and the power model. The benefit of having a central attack manager with a generic interface is that it allows new cryptosystems to be added with minimal impact to the rest of the system. When launched, the attack framework software is an interactive command-line application with intuitive help menu that allows the user to select the desired attack option along with providing necessary input parameters. The software was successfully built and tested on Microsoft Windows 7 using Microsoft Visual Studio with .NET Framework Version 4.5.

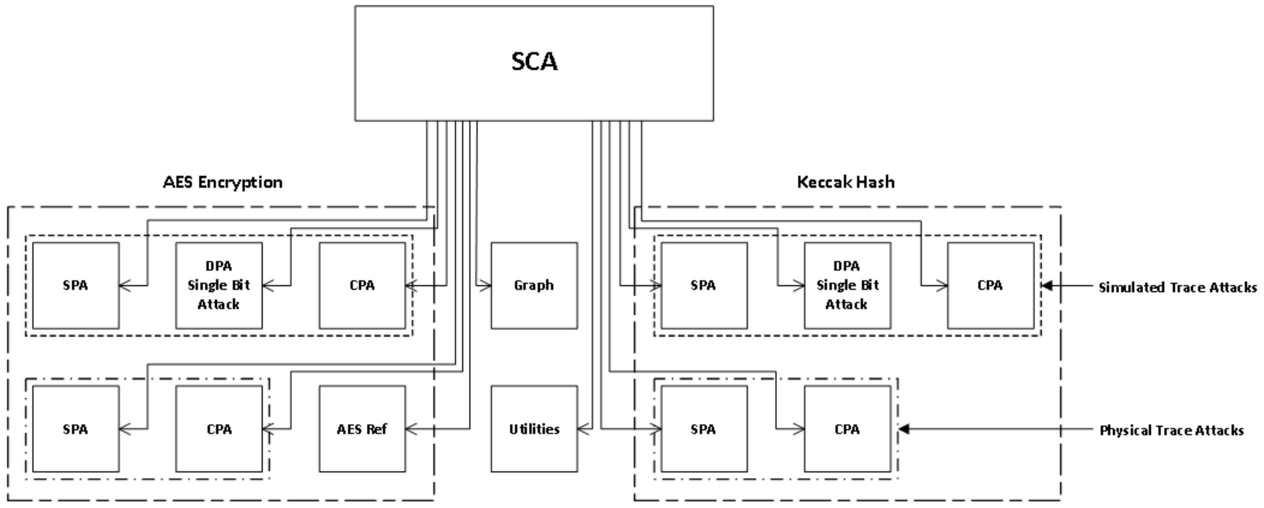


Figure 5.10: Top level class diagram of the attack framework.

5.2.1 Algorithm Design

Thus far, the discussion has been about the design of the attack framework that uses the generated power traces along with the availability of attack algorithms such SPA, DPA and CPA that could be used to attack the cryptosystem. This section will focus on the algorithm design of CPA along with the target operation of Keccak in order to come up with an accurate power model to attack MAC-Keccak successfully to retrieve the key. Based on the characteristic that the Keccak hash function absorbs the input message and processes the message with the different rounds in order to produce a message digest, one can exploit the first Keccak round operation, specifically the θ transformation, to extract the key as proposed in Figure 3.9 by Taha *et al.* Recall that the θ transformation is computed over two successive phases. The first phase is to calculate the parity of each column, which is called θ_{plane}. The second phase consists of calculating the remaining part of the θ transformation such that it computes the XOR between every bit of the state and the two parity bits of θ_{plane}. By calculating the θ_{plane} first, it speeds up the calculation of the θ transformation versus the naive way of recalculating the parity bits of the two adjacent columns for every output bit. The described steps for computing the θ transformation are illustrated in Figure 5.11. This is, indeed, how the Keccak high-speed core in Figure 4.1 implemented the θ transformation.

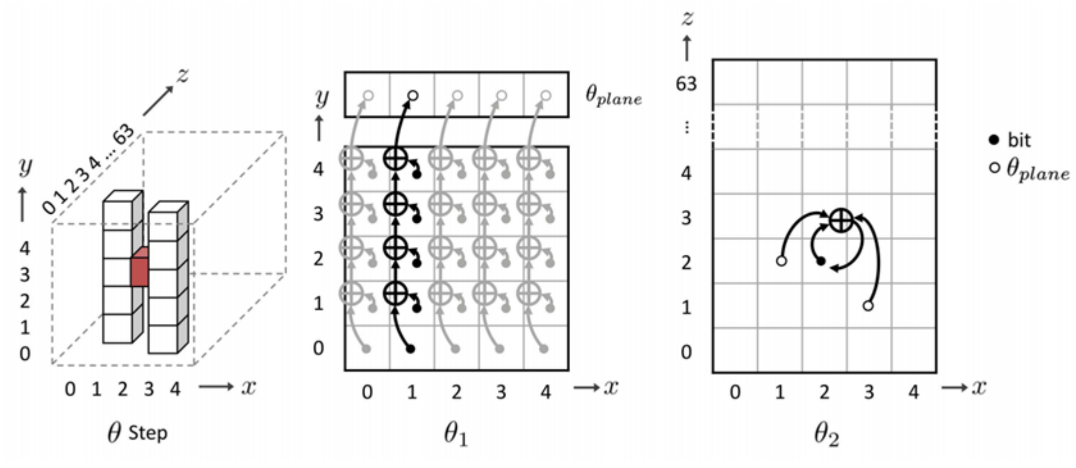
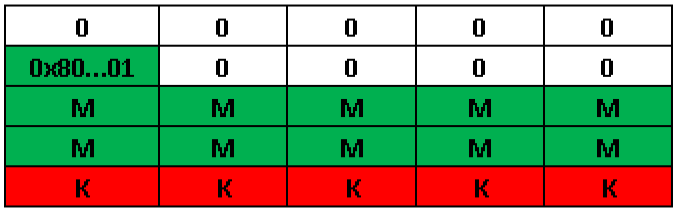


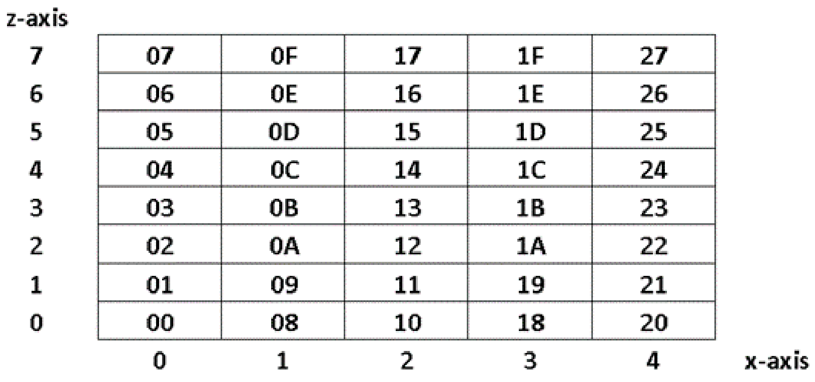
Figure 5.11: θ step, θ_1, and θ_2 [14].

According to Bertoni *et al.,* the developers of Keccak, they suggest in the Keccak reference of [2] that MAC-Keccak is secure with the key prepended to the input message. Therefore, the object of the power analysis attack of MAC-Keccak is to retrieve the prepended key of each input message. Obviously, the key is the secret part and the appended messages are public information that the attacker has access to. If the attacker is somehow able to retrieve the key, he is then able to forge the hashed messages and claim to be the valid sender. From the discussion of the Keccak high-speed core architecture in the Keccak Hardware Architecture chapter, the core has an internal state of 1600 bits, rate r of 1024 bits and capacity c of 576 bits. An external C code program is used to generate the input messages in $keccak\_in.txt$ where each message occupies 16 lanes, i.e. 1 lane of 64 bits, that is used by the test bench to feed into the core for hashing. Since the rate is 1024 bits, the padded messages generated with the prepended key were chosen to fit perfectly in 1 rate block. Therefore, Figure 5.12 and Figure 5.13 show how the message resides in the Keccak state array after absorption but before any round transformations. The key is chosen to be 40 bytes long so that it will fit perfectly on the bottom plane. Figure 5.12 shows the front view of the Keccak state for 1 message block of 1024 bits, that is, 16 lanes where the 5 key lanes occupy the bottom plane and 11 lanes of message data occupy the next three upper planes. The selection of the Key1 value is depicted in Figure 5.13 where its internal byte values increase sequentially in order to easily monitor for the correlation results and also to debug the CPA attack algorithm.



0	0	0	0	0
0x80...01	0	0	0	0
M	M	M	M	M
M	M	M	M	M
K	K	K	K	K

Figure 5.12: Front view of the Keccak state with 5 key lanes and 11 lanes of message data.



z-axis						
7	07	0F	17	1F	27	
6	06	0E	16	1E	26	
5	05	0D	15	1D	25	
4	04	0C	14	1C	24	
3	03	0B	13	1B	23	
2	02	0A	12	1A	22	
1	01	09	11	19	21	
0	00	08	10	18	20	
	0	1	2	3	4	x-axis

Figure 5.13: Top view of the Keccak state, 1st plane using Key1. Each cell is a byte.

Once we understand how the Keccak high-speed core reads in the input message, the next step is to examine the Keccak round transformations in order to come up with a meaningful target operation to perform CPA attacks. At the beginning of each hash operation, the core is reset so that the state register is cleared before each MAC-Keccak operation. Since the θ transformation of the Keccak first round has access to the key, it will serve as the point of interest for attacking. In particular, this research work focuses on the first of the two steps of θ transformation computation mentioned earlier, that is, targeting the θ_{plane}. Figure 5.14 shows the computation of θ_{plane} where the target key is the bottom plane followed by the controlled, generated messages on the next few planes. The operation of the θ_{plane} is very simple such that it only performs XOR of the bits in a given column. Therefore, the attack power model is to monitor the output bits of the θ_{plane} operation. Since everything is reset before each hash computation, monitoring the output bit of the θ_{plane} is nothing more than monitoring the number of bits set to 1. In this case, it easy to see $HW(\theta_{plane}) = HD(\theta_{plane})$ where HW stands for Hamming weight and HD is Hamming distance. Recall, the Hamming weight power model is defined such that the power consumption corresponds to the number of bits set to 1 and the Hamming distance power model is defined such that the power consumption corresponds to the number of bits that switch, i.e. 1 to 0 or 0 to 1.

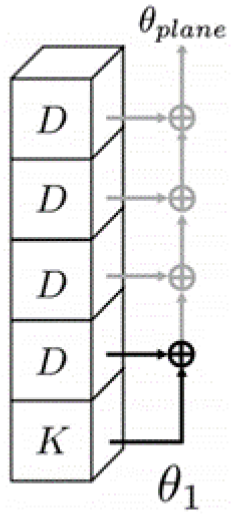


Figure 5.14: MAC-Keccak selection function target for one unknown key piece [14].

After identifying the target operation of the θ transformation for the power model to be used in CPA attacks, the next step is to apply the CPA attack that uses the generated simulated power traces described in the earlier section and the power model that is just mentioned. To understand how CPA works, Figure 5.15 shows the high-level CPA class diagram with labeled internal block components of the interaction between the power waveform class and power model class. The power waveform and power model classes with internal calculation for means and standard deviations, represent the two variables in the Pearson's correlation coefficient that can be seen in Equation (2.12). The CPA class diagram also shows the current data structures that are being used to store the read-in power traces along with the computed power model. The objective of creating this CPA class diagram is to make and facilitate the understanding of how CPA works, intuitively. One important thing to note from the power waveform class is that it takes the power out files with power values recorded at 10ps resolution and averages the power values for every 1ns to elevate power values versus ps time domain to power values versus ns time domain. The purpose of this is to save space in memory for the data structure to store processed power traces in the power waveform class.

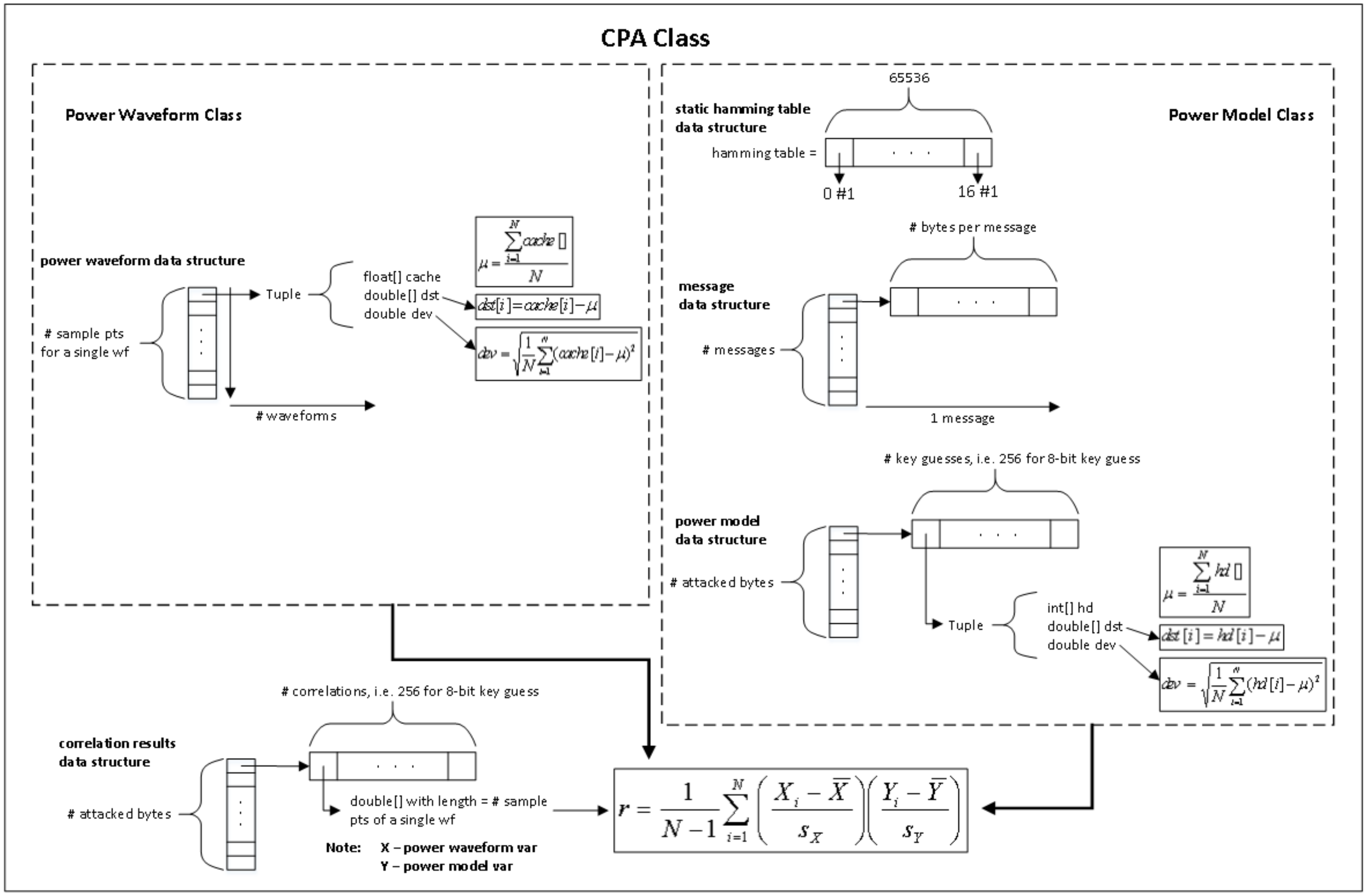

CPA Class
Power Waveform Class
power waveform data structure
# sample pts for a single wf
# waveforms
Tuple
float[] cache
double[] dst
double dev
$\mu = \frac{\sum_{i=1}^{N} cache[]}{N}$
$dst[i] = cache[i] - \mu$
$dev = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(cache[i] - \mu)^2}$
Power Model Class
static hamming table data structure
65536
hamming table =
0 #1
16 #1
message data structure
# bytes per message
# messages
1 message
power model data structure
# key guesses, i.e. 256 for 8-bit key guess
# attacked bytes
Tuple
int[] hd
double[] dst
double dev
$\mu = \frac{\sum_{i=1}^{N} hd[]}{N}$
$dst[i] = hd[i] - \mu$
$dev = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(hd[i] - \mu)^2}$
correlation results data structure
# correlations, i.e. 256 for 8-bit key guess
# attacked bytes
double[] with length = # sample pts of a single wf
$r = \frac{1}{N-1}\sum_{i=1}^{N}\left(\frac{X_i - \overline{X}}{s_X}\right)\left(\frac{Y_i - \overline{Y}}{s_Y}\right)$
Note: X – power waveform var
Y – power model var

Figure 5.15: Correlation power analysis overview.

The usage of the SPA is effective in cracking RSA because it exploits the square-and-multiply operation of the algorithm. However, using SPA to retrieve the key from MAC-Keccak is not sufficient. Like any tool, it is used as a starting point to extract preliminary information that a more sophisticated tool like CPA could use. Figure 5.16 shows the simple power analysis of power trace #007 using Key1 with 24 rounds of Keccak. The curve shows the characteristics that are mentioned in Keccak Hardware Architecture such that the first 16 spikes correspond to reading the 16 input lanes of a single block message from a file, followed by 24 spikes for the 24 rounds, 3 small spikes from waiting for the valid data out, and finally 4 additional spikes from writing the result to the output file. Any spikes following after that result from reseting the hash operation. The advantage of applying SPA to MAC-Keccak is to see at which clock cycle or point in time each Keccak operation starts and ends. The simulated power traces in this work use a clock with a 20ns cycle.

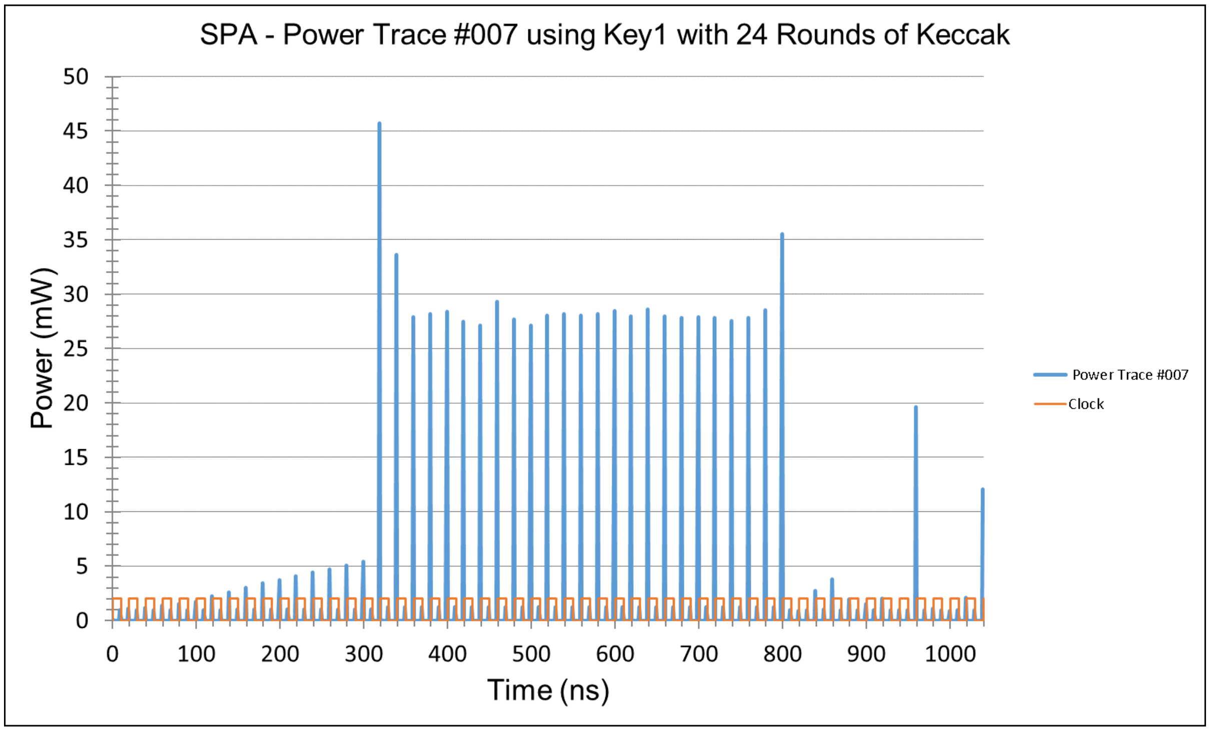


Figure 5.16: Simple power analysis of power trace #007 using Key1 with 24 rounds of Keccak.

Figure 5.17 shows the zoomed-in power trace using the first 16 clock cycles of reading the input message followed by the two clock cycles for 2 Keccak round operations. This figure clearly shows a high power spike at time 320ns which is the first operation and that is the point of interest for the power attack. Since the time scale for both SPA plots is in ns, the number of samples per power trace for using 24 rounds of Keccak is 1040 sample points and 360 points for 2 rounds of Keccak. The number of sample points used plays a critical role in the attack performance between using the two different Keccak rounds and that will be discussed in the next chapter, Results and Analysis. One could even go further to restrict the number of power samples per waveform up to and including the first round and not necessarily to include the second round. The reason this work did not choose to do that is because it is better to view the waveform correlation results with at least one other Keccak round besides the first round in order to verify the selected target operation of θ_{plane} really produces a useful power model.

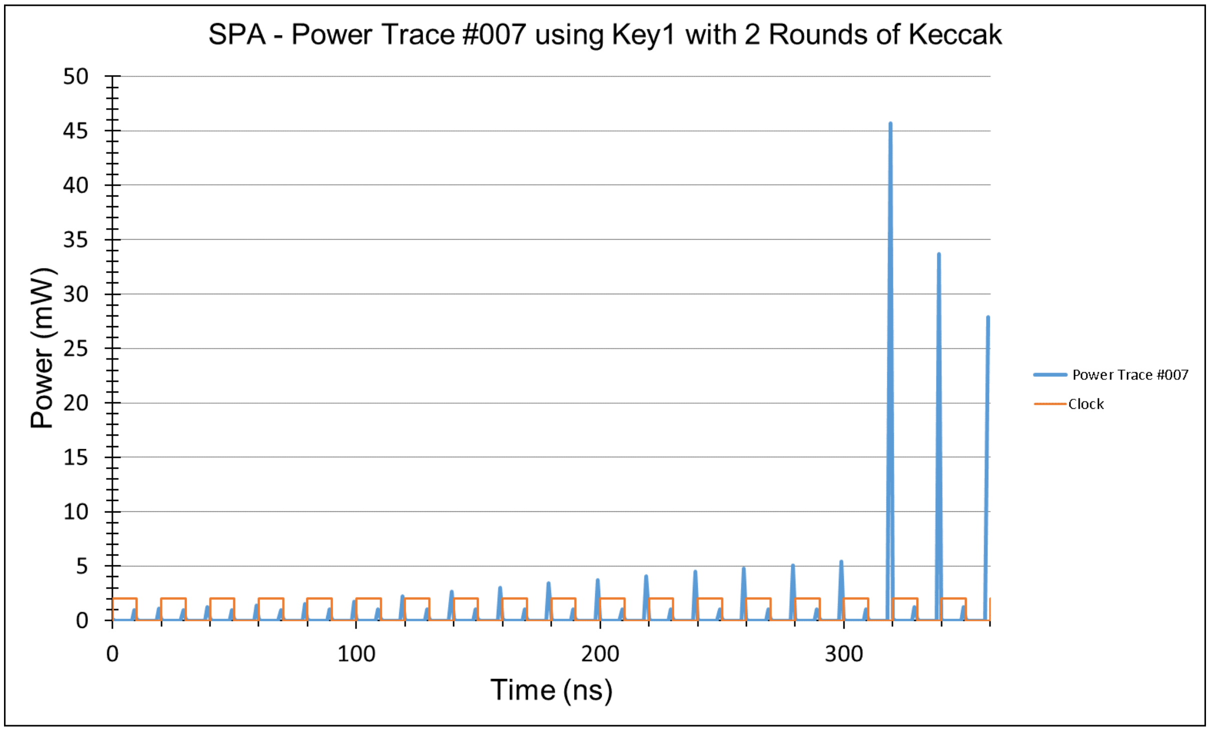
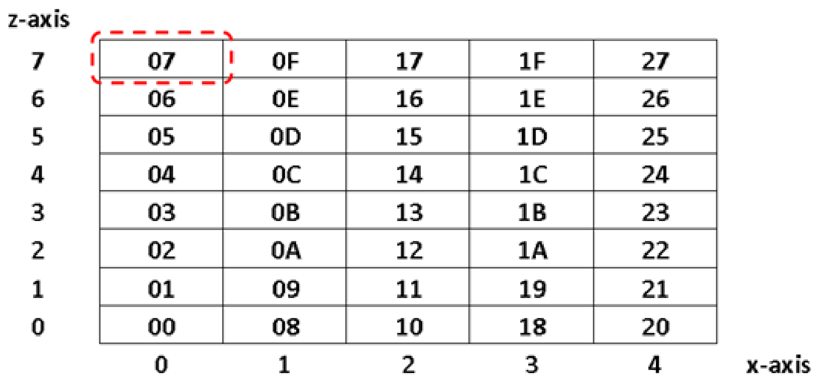


Figure 5.17: Simple power analysis of power trace #007 using Key1 with 2 rounds of Keccak.

Using the knowledge of SPA to extract the MAC-Keccak power waveform characteristics such as when each round starts and ends, everything that is needed for a CPA attack against MAC-Keccak is ready. Figure 5.18 shows the top view of the Keccak state and the attack target of 1 key byte of Key1 on the 1st plane encircled by the red dotted line denoted as partial key piece 7.



z-axis					
7	07	0F	17	1F	27
6	06	0E	16	1E	26
5	05	0D	15	1D	25
4	04	0C	14	1C	24
3	03	0B	13	1B	23
2	02	0A	12	1A	22
1	01	09	11	19	21
0	00	08	10	18	20
	0	1	2	3	4

Figure 5.18: Top view of the Keccak state, attack target of 1 key byte of Key1 on the 1st plane. Each cell is a byte.

The CPA attack uses the 8-bit selection function key guess size to carry out the attack against the targeted partial key piece 7 utilizing 24 rounds of Keccak and 2 rounds of Keccak in Figure 5.19 and Figure 5.20, respectively. Observing these two curves with 20K power traces applied among the different key guesses, the correct key guess does not stand out at all. Using an 8-bit CPA selection function key guess size should produce 256 correlation traces where each trace corresponds to a key guess. The figures only show the first 16 correlation traces as one knows the correct key guess is 0x07, therefore it is sufficient to only show the first 16 correlation traces. The clock signal is shown as the orange curve. There seems to be a high correlation at time 320ns which is the Keccak first-round operation.

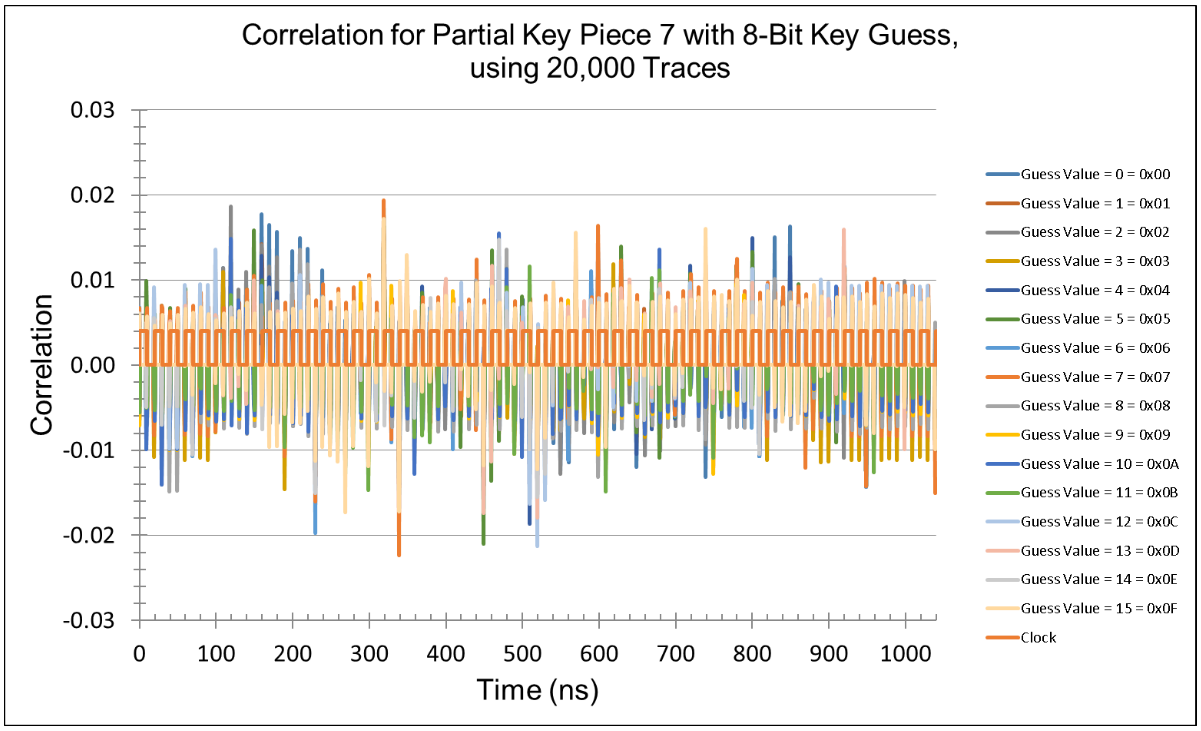


Figure 5.19: CPA time domain - 24 rounds of Keccak using 20K traces with Key1.

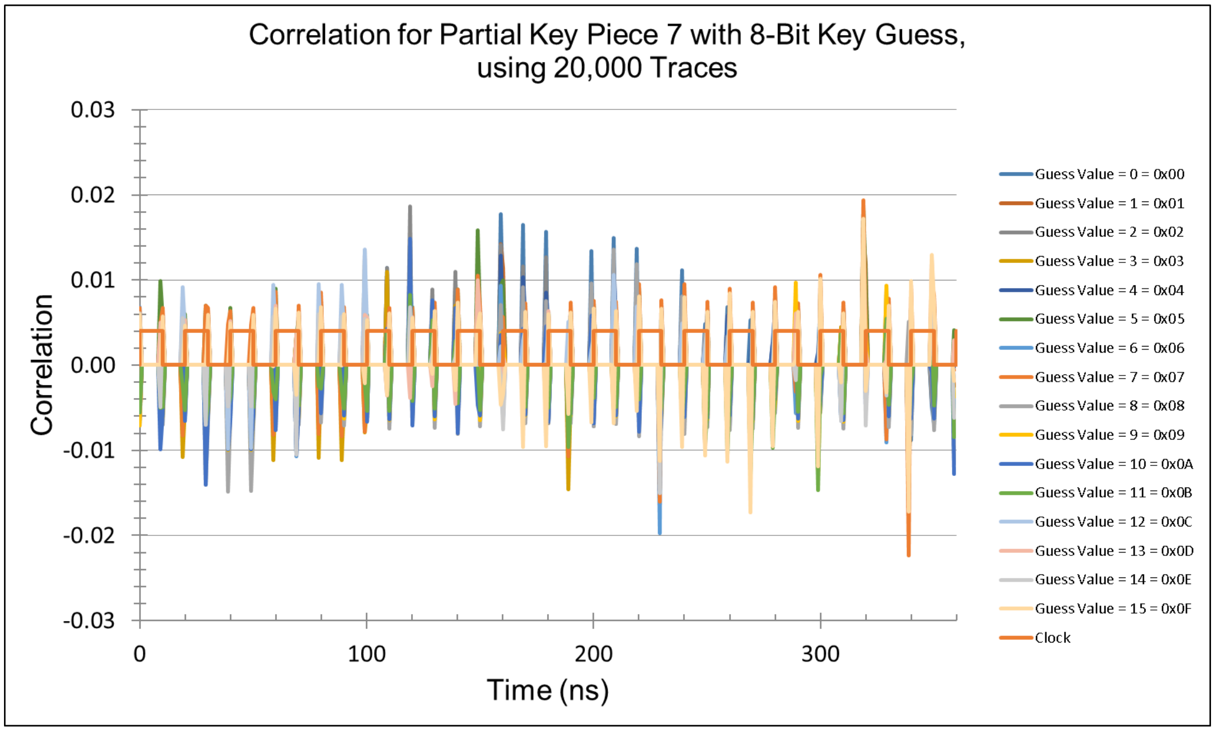


Figure 5.20: CPA time domain - 2 rounds of Keccak using 20K traces with Key1.

Figure 5.21 and Figure 5.22 show a side-by-side comparison of the correlation waveform for the ‘correct’ key byte guess that is 0x07 and the wrong key byte guess, chosen to be 0x0B, for CPA using 24 rounds of Keccak. The reason why key guess 0x0B is chosen to be compared with the supposed correct key byte 0x07 is that both have the same number of 1s. The purpose is to see how having the same number of bit 1s set with different key guesses impacts the correlation waveforms. It is interesting to see that at time 320ns, which is the beginning of the first round, there is a positive correlation value spike but following that, there is a negative spike with a much larger magnitude. Clearly, looking at these plots, there is no distinct correlation between the key guess and the applied power traces.

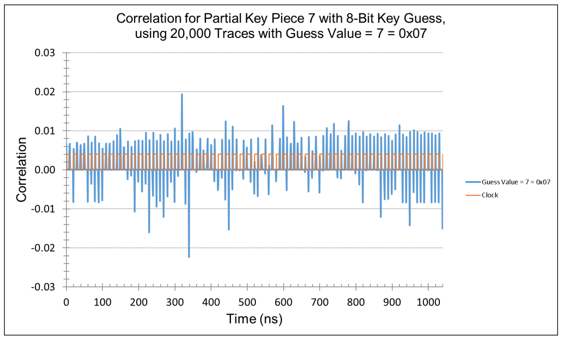


Figure 5.21: CPA time domain - examining the supposed “correct” key byte guess value for 24 rounds of Keccak using 20K traces that used Key1.

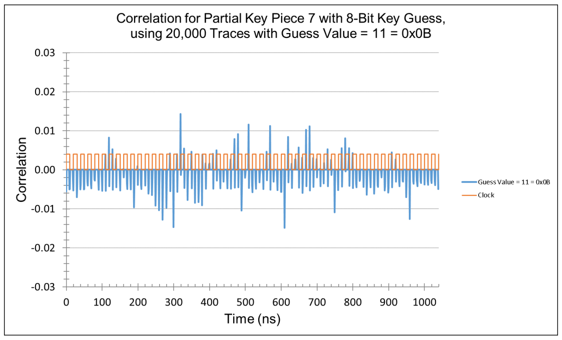


Figure 5.22: CPA time domain - examining the wrong key byte guess value for 24 rounds of Keccak using 20K traces that used Key1.

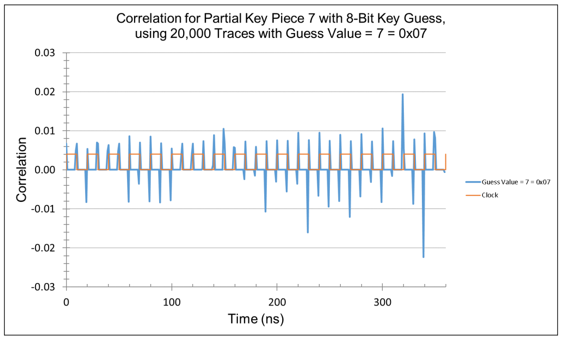


Figure 5.23: CPA time domain - examining the supposed “correct” key byte guess value for 2 rounds of Keccak using 20K traces that used Key1.

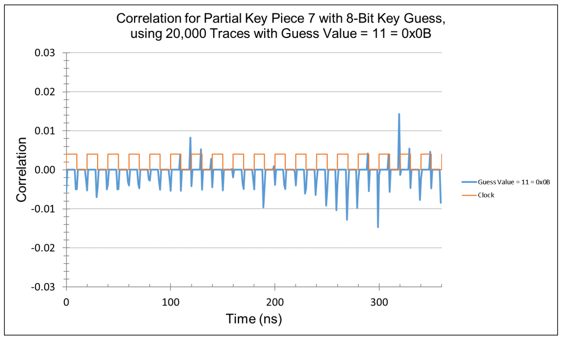


Figure 5.24: CPA time domain - examining the wrong key byte guess value for 2 rounds of Keccak using 20K traces that used Key1.

Figure 5.23 and Figure 5.24 show the same analyzed key guesses as the previous figures but these curves were generated from the CPA attacks focusing on the first 2 rounds. Both figures show there is a large positive spike at time 320ns among the other time points of the correlation waveform and one might jump to the conclusion that this is the correct key byte. Therefore, to prevent this from happening, the CPA algorithm also generates the correlation versus key guess value graph where it plots the highest correlation value of each key guess as can be seen in Figure 5.25 and Figure 5.26 for the usage of 24 rounds and 2 rounds respectively. The CPA attack program then searches among the key guesses for their highest correlation values and whichever key guess provides the highest correlation value is the best candidate for the target key piece. Please notice the usage of the term 'key piece' since the CPA application could use 1-bit, 2-bit, 4-bit, 8-bit or even 16-bit selection function key guess sizes. This section focuses specifically on the usage of the 8-bit CPA selection function key guess size for ease of discussion and understanding. The chapter on 'Results and Analysis' will elaborate more on the usage and experiment results of the n-bit CPA selection function key guess size. Using the term 'key piece' removes specific dependency for using or referring to the 8-bit CPA selection function key guess size. Where used, key byte is understood to be using the 8-bit CPA selection function key guess size. For an 8-bit CPA selection function key guess size there are 256 key guesses, 1-bit CPA has 2 key guesses, 2-bit CPA has 4 key guesses, 4-bit CPA has 16 key guesses and 16-bit CPA has 65,536 key guesses. Intuitively, one would think the two plots of highest correlation value of each key guess would be the same for the usage of 24 and 2 rounds of Keccak since the point of interest for attack is the first round. The reason for this difference is that when using fewer power traces, the correlation contributions of the rounds beyond the first two rounds are larger than the first round. In other words, the correlation values from round 2 to 24 provide false positive results. This is the reason for the CPA attack, in particular the power waveform class, to extract only the beginning of the power waveforms up to the first 2 rounds and discard the rest. It is interesting to see the magnitude of the supposed correct key guess 0x07 is nowhere near the top potential candidate for the CPA algorithm to consider.

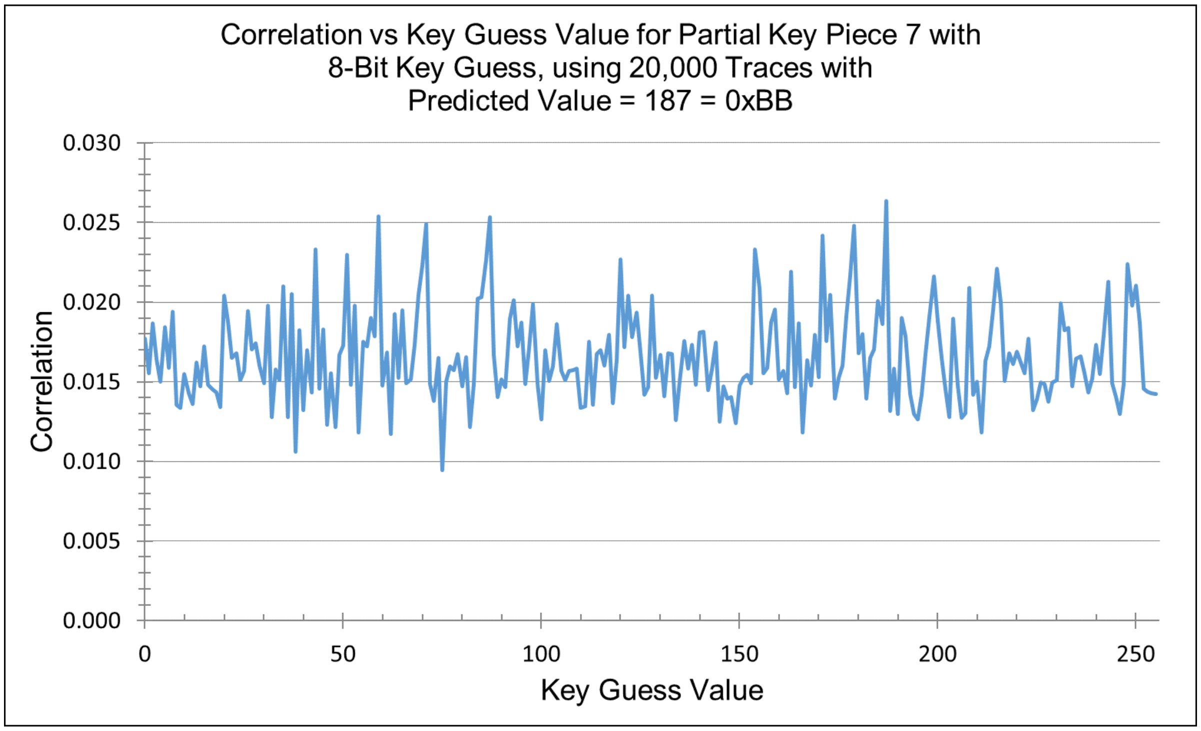


Figure 5.25: CPA predicted the wrong key value for Key Piece 7 using 24 rounds of Keccak.

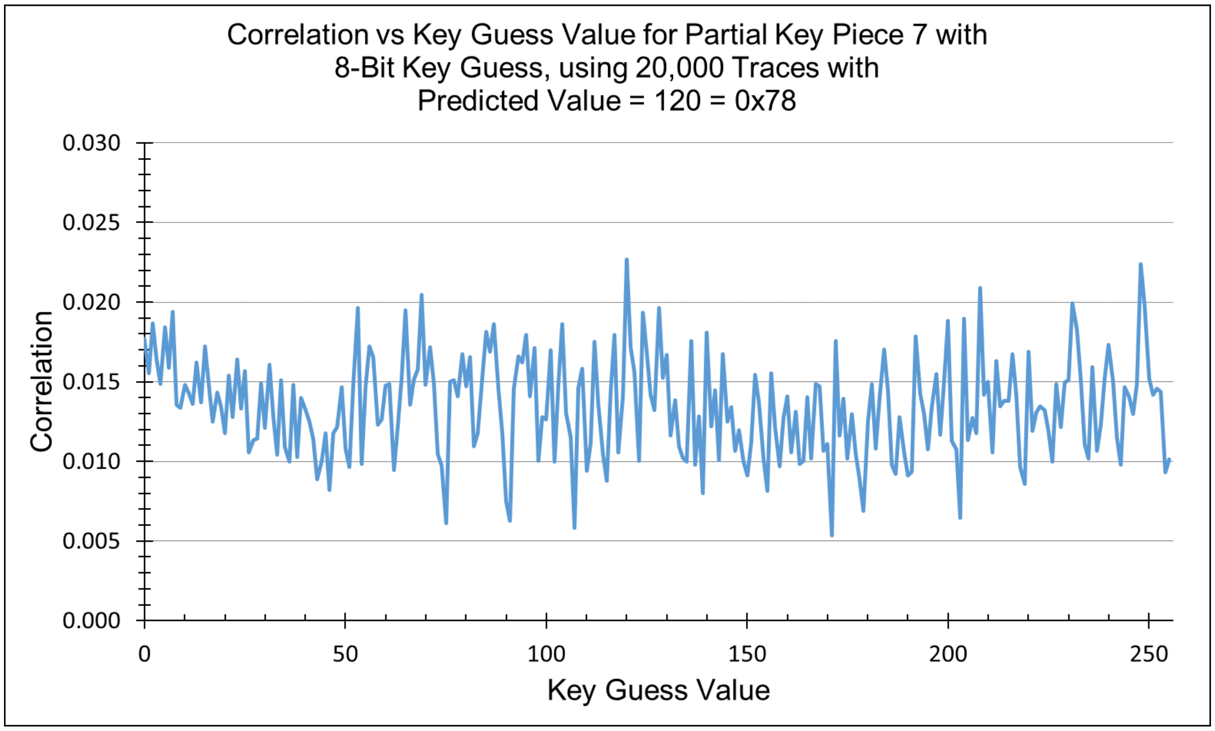


Figure 5.26: CPA predicted the wrong key value for Key Piece 7 using 2 rounds of Keccak.

From the two previous plots of the correlation versus key guess value for the target partial key piece 7, it seems there is no hope for the CPA attack to extract the key from MAC-Keccak using the exploitation of the θ_{plane} operation in the Keccak first round. With past experience of attacking AES using DPA and CPA algorithms, the usage of more power traces led to better results for extracting the correct key guess. This is true when increasing from 20K applied traces to 200K traces for the CPA attack against MAC-Keccak and is able to extract the target key byte correctly. Both Figure 5.27 and Figure 5.28 for the usage of 24 and 2 Keccak, rounds respectively, show the highest positive correlation value at time 320ns and that is associated with key guess 0x07. All of the other correlation values associated with the other time points hover around the x-axis. The CPA time domain plot for the 24 rounds clearly shows there is a high correlation at the beginning of round 1 that is at 320ns between the 200K power traces and the power model of the targeted θ_{plane} operation.

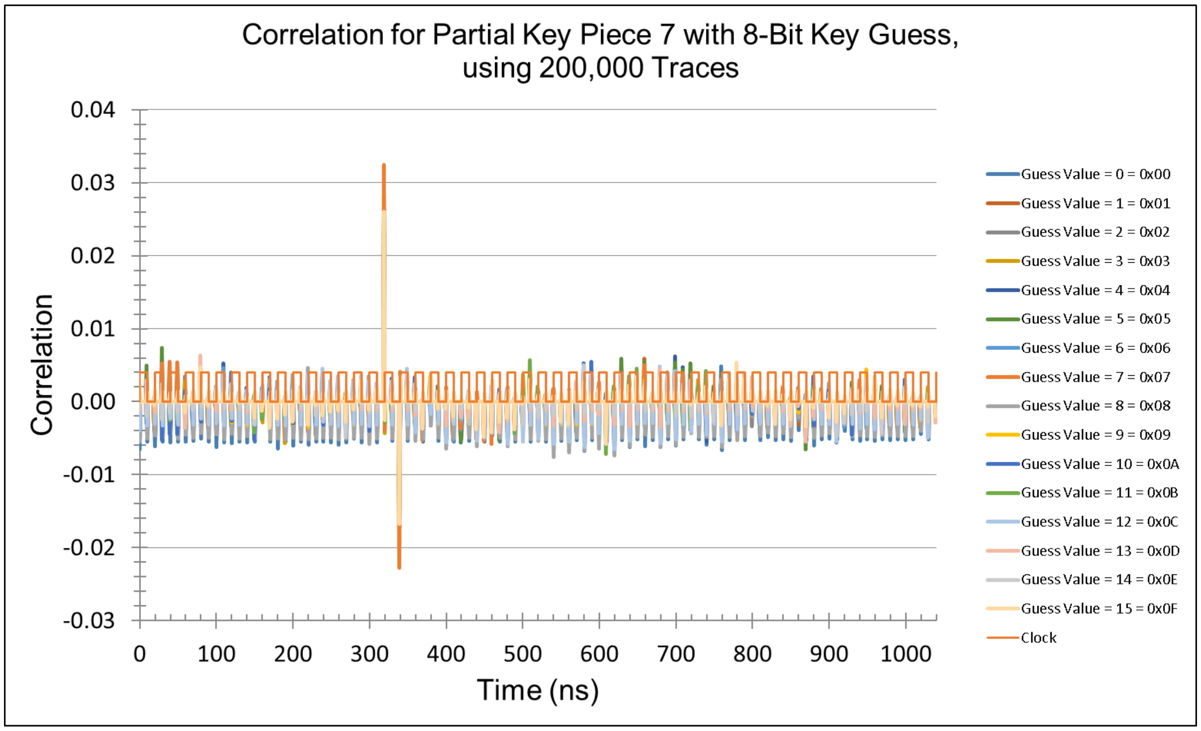


Figure 5.27: CPA time domain - 24 rounds of Keccak using 200K traces with Key1.

Figure 5.28 shows the CPA time-domain plot for 2 rounds of Keccak using 200K traces with Key1. Unlike the previous plot that shows the CPA time domain across 24 rounds of Keccak where it is difficult to see the exact time associated with the highest correlation value, in this figure, it is more evident the high positive spike is associated with the first round Keccak operation. The next highest spike in terms of magnitude in these plots is right at the beginning of the second round computation which is 340ns. It is critical to look for positive correlation values in these graphs as it indicates a positive relationship between the two variables which are the collected simulated power traces and the power model for a target operation. Therefore, one should not take the magnitude of the correlation results as an indication of the correct key guess as they will provide false positive results but leave the correlation results the way it is and then look for the highest positive correlation value.

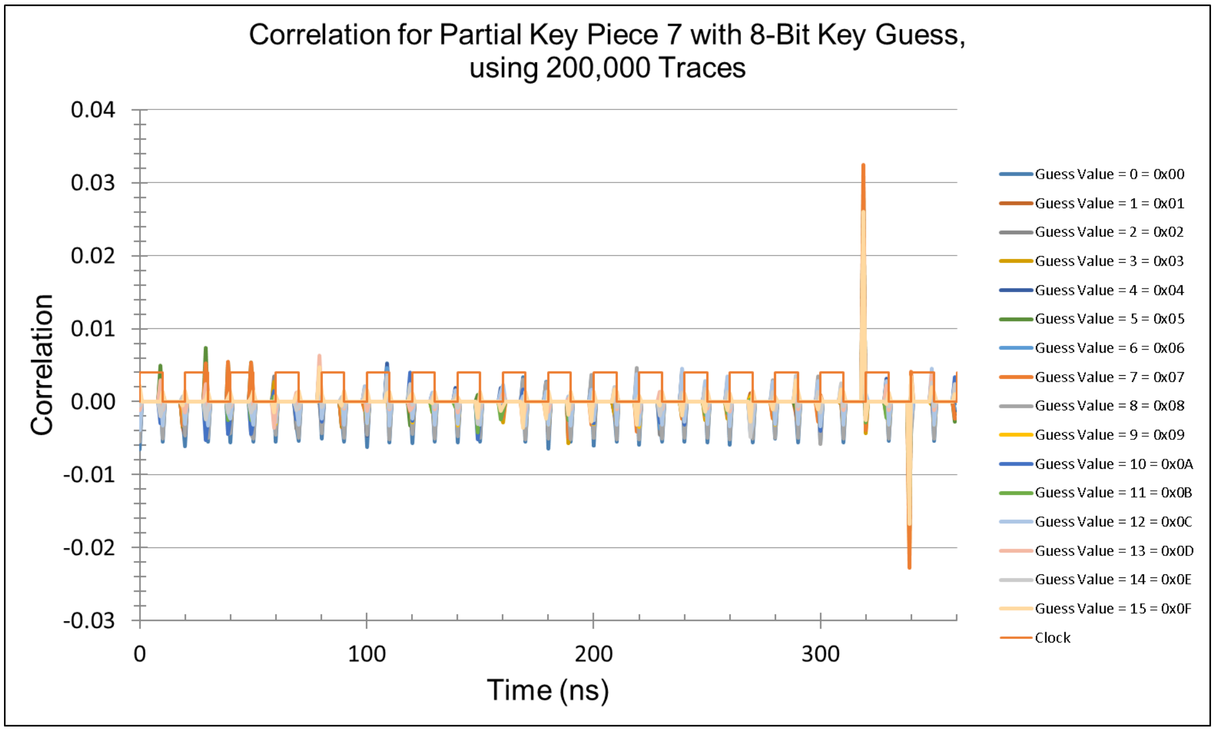


Figure 5.28: CPA time domain - 2 rounds of Keccak using 200K traces with Key1.

The previous plots show the superposition of the first 16 different key guess CPA time domain curves, and it is hard to see the curve of the correct key byte other than a huge spike at the time point 320ns. Figure 5.29 and Figure 5.30 show a single curve of the CPA time domain for the correct key byte guess and an incorrect key byte as a result of the CPA attacking on all 24 rounds. Unlike the earlier plots that show the same information but from the application of 20K traces, this time it is more obvious that the true correct key guess has the highest correlation spike. For the wrong key guess in Figure 5.30 with the key value 0x0B, there is still a high correlation value at time 320ns among its time points but its magnitude is nothing compared to the magnitude of the correct key byte guess in Figure 5.29. The reason why key guess 0x0B has a similar CPA time domain curve as the correct key guess is the commonality of the number of 1 bits in the 8-bit wide key guess size. It is interesting to see the correlation computation of the CPA algorithm successfully differentiate the order of 1s in the correct key guess versus the incorrect key guess even though both key guesses share the same number of 1s.

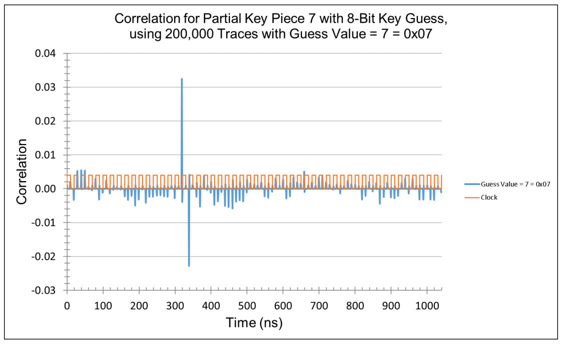


Figure 5.29: CPA time domain - examining the correct key byte guess value for 24 rounds of Keccak using 200K traces that used Key1.

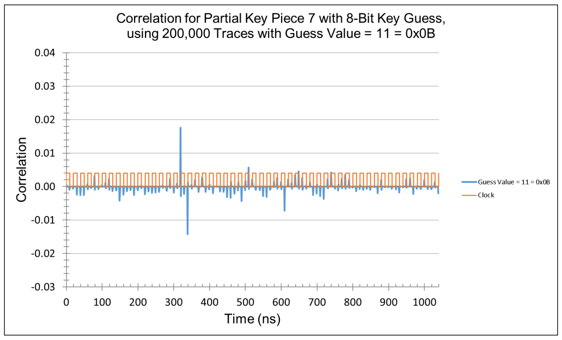


Figure 5.30: CPA time domain - examining the wrong key byte guess value for 24 rounds of Keccak using 200K traces that used Key1.

Figure 5.31 and Figure 5.32 show single curves of the CPA time domain for the correct key byte guess and the incorrect key byte as a result of the CPA attacking the first 2 rounds of Keccak. The interesting thing to observe from these CPA time domain plots is to see, close up, the correlation value activities over time. For example, in both figures, there is a clear sign of high correlation at the beginning of the first round of Keccak and fairly high negative correlation at the beginning of the second round. What is interesting is that for the application of 200K power traces, each time point of the applied traces that does not correlate to the power model of the target operation hugs tightly to the x-axis. It makes sense for this to happen. As more samples of the power traces are applied, the calculation of the correlation computation is more accurate at reflecting the relationship between the simulated power traces and the power model. Plus, the usage of more power traces removes any outlier power values in simulated traces and removes measurement or environment noise captured in the physical traces.

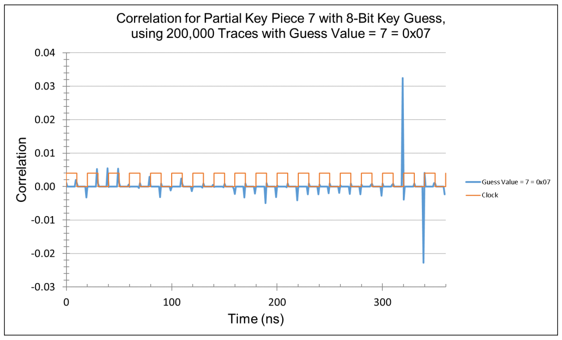


Figure 5.31: CPA time domain - examining the correct key byte guess value for 2 rounds of Keccak using 200K traces that used Key1.

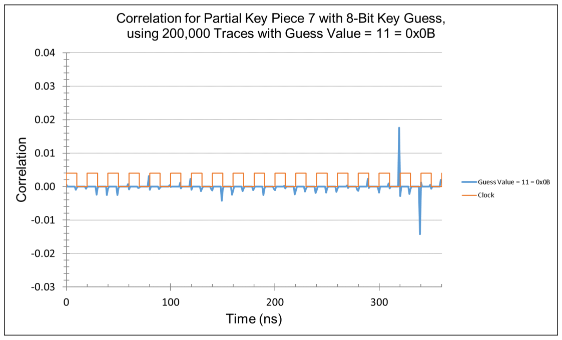


Figure 5.32: CPA time domain - examining the wrong key byte guess value for 2 rounds of Keccak using 200K traces that used Key1.

As discussed before, the CPA attack application provides the correlation versus key guess value plot where the highest correlation value of each key guess is plotted in the graph upon the completion of the attack. This graph can be seen in Figure 5.33 and Figure 5.34 for the usage of 24 rounds and 2 rounds, respectively. Unlike the earlier plots for the correlation versus key guess value as the result of the 24 and 2 round attacks using 20K traces where the correlation value of the supposed correct key guess is nowhere near the top, the correlation of the correct key guess in these figures as a result of using 200K traces stand out among the correlation values of different key guesses. At a first glance of the two figures, it might seem they are identical but they are not. The graphs are slightly different for key guesses with very low correlation values such as from key guess 150 to 160. The slight difference is a result of the false positive correlation values provided by the rounds after the first 2 rounds. By using a larger number of power traces to minimize this problem and the fact the two curves share common characteristics overall, especially no differences for key guesses with correlation values, it is safe to use either 24 or 2 rounds of Keccak to attack MAC-Keccak providing the attacker uses a sufficient number of power traces. This anomaly can be seen with the success rate plots using 8K trace steps in the CPA algorithm to recover 8 and 40 bytes of Key1 in the next chapter, Results and Analysis.

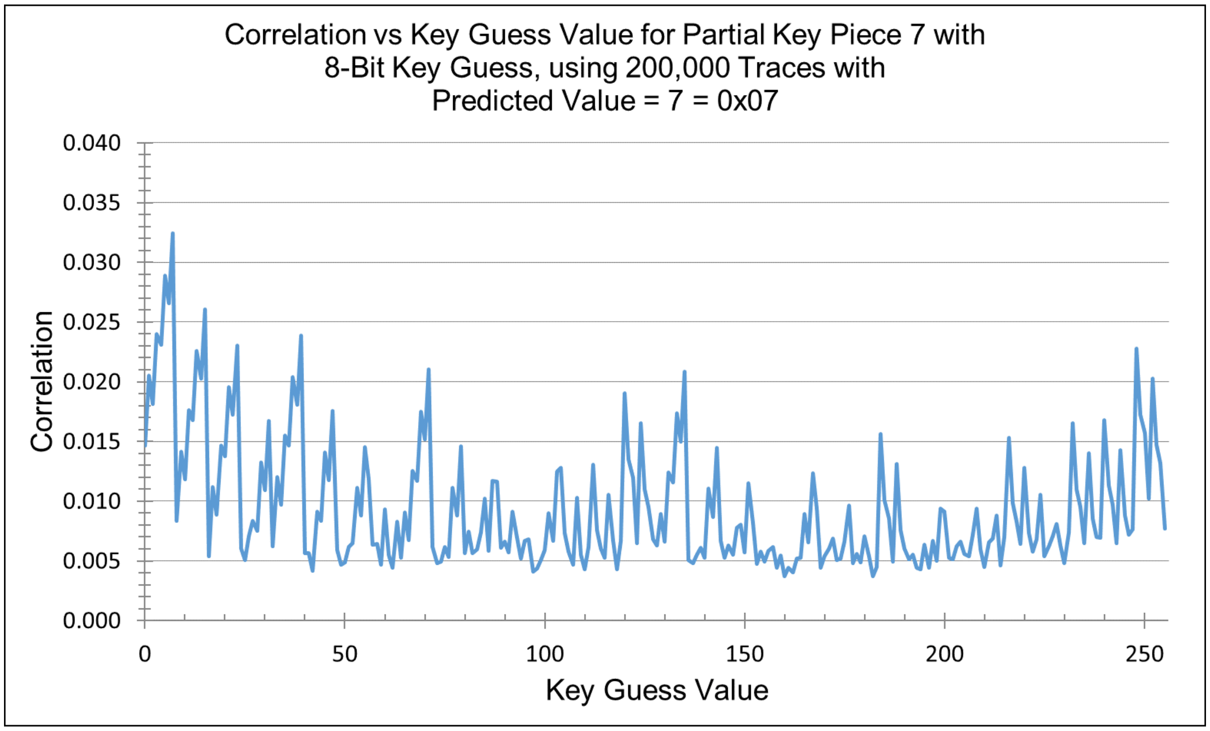


Figure 5.33: CPA predicted the correct key value for Key Piece 7 using 24 rounds of Keccak.

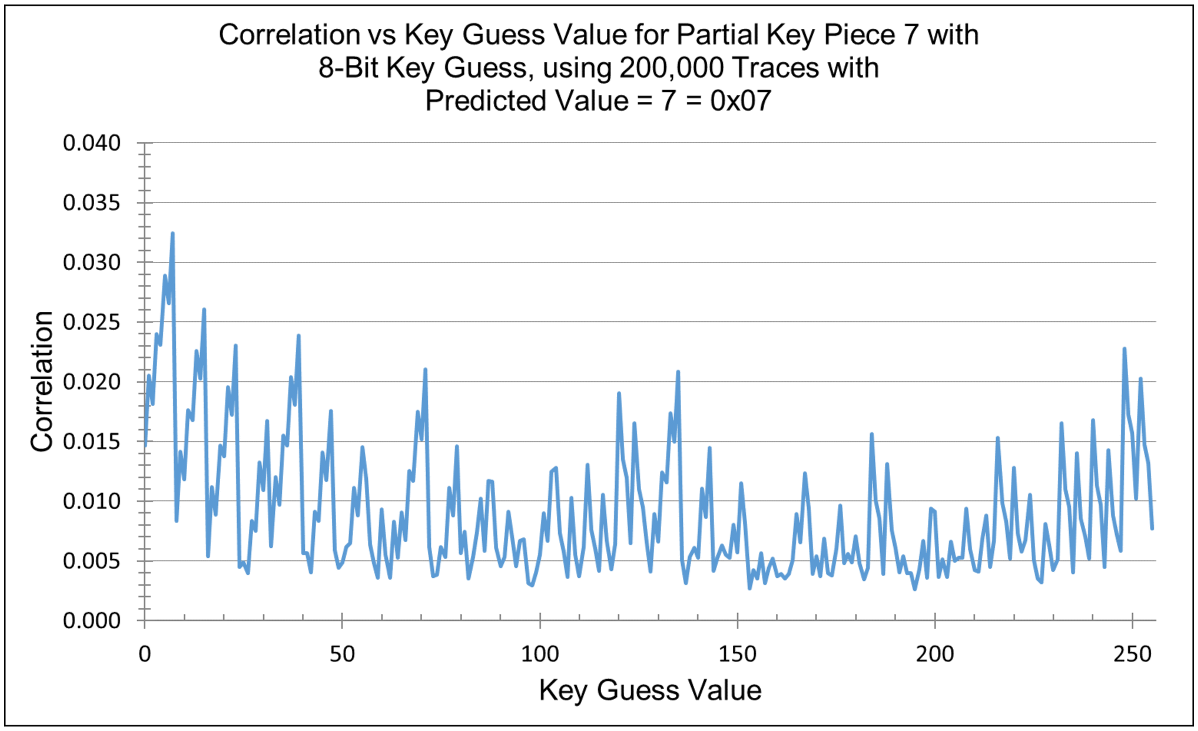
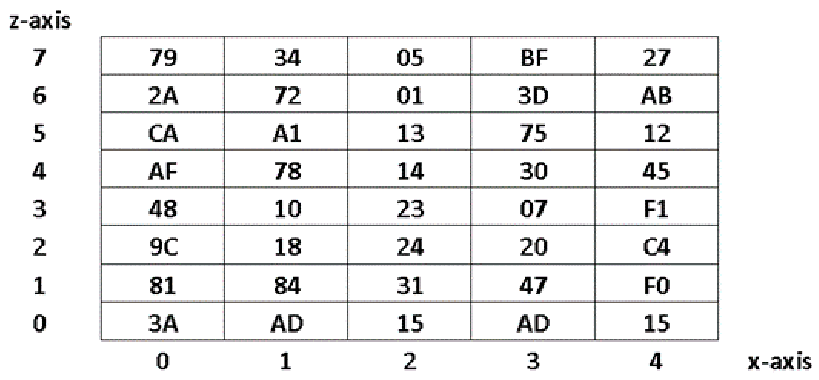


Figure 5.34: CPA predicted the correct key value for Key Piece 7 using 2 rounds of Keccak.

Now that the CPA program along with the power model for targeting the θ_{plane} of the first Keccak round are proven to be useful and effective for attacking a single key byte of Key1, the next step is to see how effective it is to attack a single key lane when the bottom plane of the Keccak state is filled with the whole key, i.e. 5 key lanes or equivalently 40 key bytes. Another way to verify the correctness of the CPA algorithm along with the chosen power model for targeting the θ_{plane}, is to attack a completely different key where each key byte has a random value rather than each key byte having an incremental value as in the case of Key1. The suggested 40 byte long key, Key2, with random value for each key byte is illustrated in Figure 5.34 where the key takes up the whole bottom plane of the Keccak state array. The results of these CPA attack experiments will be discussed and analyzed in the Results and Analysis chapter.



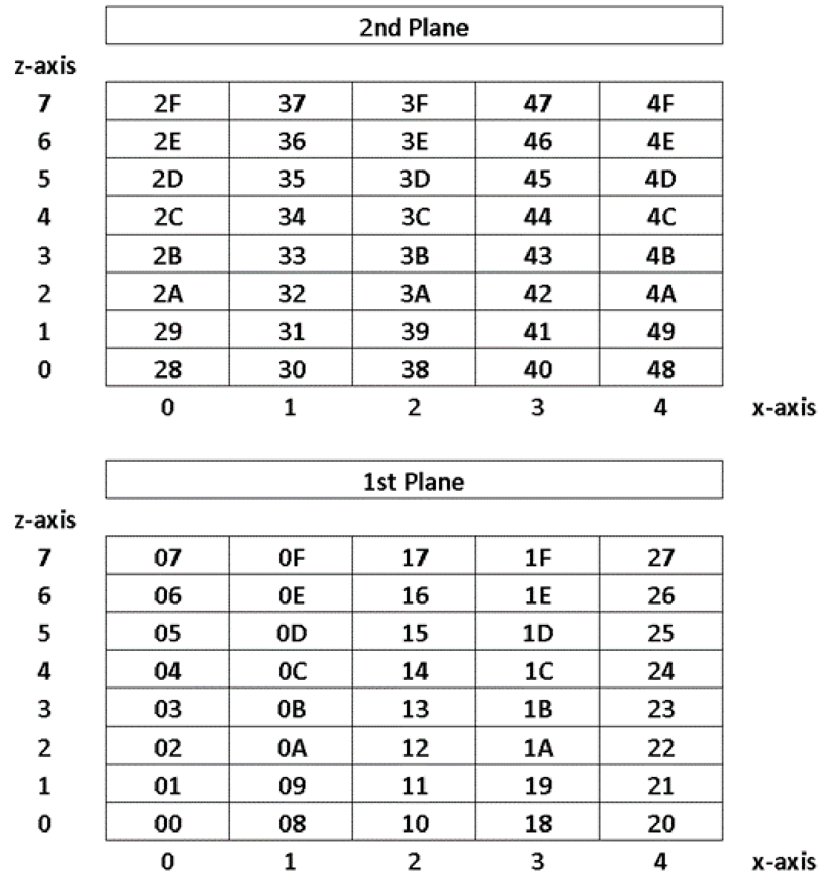
z-axis						
7	79	34	05	BF	27	
6	2A	72	01	3D	AB	
5	CA	A1	13	75	12	
4	AF	78	14	30	45	
3	48	10	23	07	F1	
2	9C	18	24	20	C4	
1	81	84	31	47	F0	
0	3A	AD	15	AD	15	
	0	1	2	3	4	x-axis

Figure 5.35: Top view of the Keccak state, 1st plane using Key2. Each cell is a byte.

Since a key that is 40 bytes long only fills up the bottom plane of the Keccak state array, it is with high confidence the implemented CPA algorithm with the power model targeting the θ_{plane} of the Keccak first round will be able to recover either all or most of the key bytes. Recall from the Previous Work chapter that Taha and Schaumont in [14] focus on attacks of software implementations of MAC-Keccak to retrieve the secret key of various key lengths and some are beyond 1 Keccak plane. Attacking software implementations of MAC-Keccak is different from hardware implementations. Pei *et al.* in [15] only perform attack on a hardware implementation of MAC-Keccak that utilizes a key that fits the Keccak bottom plane, perfectly. Thus this research is the first work to apply side channel vulnerability assessment of the hardware implementation using gate-level circuit-simulated power consumption waveforms from Synopsys EDA tools, in particular, using the 130-nm CMOS standard cell library, and also the first to explore and analyze method of attacking key expansion into the second plane of the Keccak state array for the hardware implementation. The chosen key for this experiment to spand the full 2 key planes is the extension of Key1 with an additional 40 key bytes where each key byte has an incremental value. The layout of the key in the Keccak state array from the front view of the Keccak state is depicted in Figure 5.36 where there are 10 key lanes considered as 80 unknown bytes. In this scenario, the attacker has access to 6 lanes (48 bytes) of controlled messages. Figure 5.37 shows visualization of the key from the top view of the Keccak state. For reasons similar to why Key1 was chosen to have sequential values for each key byte, the key for this experiment is chosen for ease of monitoring the correlation results and debugging the CPA attack algorithm.

0	0	0	0	0
0x80...01	0	0	0	0
M	M	M	M	M
K	K	K	K	K
K	K	K	K	K

Figure 5.36: Front view of the Keccak state with 10 key lanes and 6 lanes of message data.



2nd Plane

z-axis					
7	2F	37	3F	47	4F
6	2E	36	3E	46	4E
5	2D	35	3D	45	4D
4	2C	34	3C	44	4C
3	2B	33	3B	43	4B
2	2A	32	3A	42	4A
1	29	31	39	41	49
0	28	30	38	40	48
x-axis	0	1	2	3	4

1st Plane

z-axis					
7	07	0F	17	1F	27
6	06	0E	16	1E	26
5	05	0D	15	1D	25
4	04	0C	14	1C	24
3	03	0B	13	1B	23
2	02	0A	12	1A	22
1	01	09	11	19	21
0	00	08	10	18	20
x-axis	0	1	2	3	4

Figure 5.37: Top view of the Keccak state, 1st and 2nd planes. Each cell is a byte.

The proposed method for attacking the 2 key planes of the Keccak state array is to use the same power model for targeting the θ_{plane} operation of the θ transformation of Keccak first round. The same reason stands as before for targeting θ_{plane}: its access to the key before the key is diffused into the different parts of the Keccak state from the different transformations. Figure 5.38 shows the MAC-Keccak selection function target for two unknown key pieces where if one is to apply the same CPA attack method for targeting the θ_{plane}, it will only get the parity of the unknown key pieces. By retrieving the parity of the two unknown key pieces, the information does not help in any way to extract or retrieve the two key pieces. Therefore, the proposed process to extract the two unknown key pieces is to apply n-bit x n-bit cascade CPA attack using the same power model to target the θ_{plane}. Think of the n-bit x n-bit cascade θ_{plane} attack as a key guess, not across the z plane in the normal case, but across the y plane. For example, a 4-bit x 4-bit cascade CPA θ_{plane} attack produces 256 key guesses where the first plane 4-bit selection provides 16 guesses, the second plane 4-bit selection provides 16 guesses and that is a permutation of 256 guesses. The same concept can be applied to 8-bit x 8-bit cascade CPA θ_{plane} attack to produce 65,536 key guesses. Intuitively, based on the analysis of the key guesses the CPA algorithm has to go through between 4-bit x 4-bit cascade CPA θ_{plane} attack and 8-bit x 8-bit cascade CPA θ_{plane} attack, 4-bit x 4-bit cascade CPA attack will perform much faster but suffer from lower SNR than 8-bit x 8-bit cascade CPA attack. Before performing n-bit x n-bit cascade CPA attack, there needs to be some way to confirm power analysis attack is doable for two key lanes given the fact the two key planes are static across different power simulations and only the 6 remaining lanes can be controlled by the attacker. The concern here is if the 6 message lanes provide enough randomness such that the output of the θ_{plane} is still meaningful. Recall in the early case where the key spans only the first plane and there are 11 lanes of controlled messages. The 11 lanes of controlled messages mean the messages occupy at least 2 full planes of the Keccak state array which provides for more randomness that is beneficial in the power model mean and standard deviation calculations to filter between the wrong and right key guess. This may not be the case for attacking a key that occupies two planes, leaving room for only 1 full plain of controlled message. What has just discussed is the theory attack that may not be the case when put in practice. To test this theory, there are three experiments that are done in an attempt to extract the key. Experiment 1 attempts to extract the key on the bottom plane. Experiment 2 attempts to extract the key on the second plane. Finally, experiment 3 is for performing n-bit x n-bit cascade CPA attack. The same set of simulated power traces can be used for the three experiments and the power waveform does not have to be changed at all to accommodate the test of these experiments. However, the power model class has to be tweaked slightly for each experiment. Since the attack framework is written in a modular fashion, it is not hard to accommodate these test experiments.

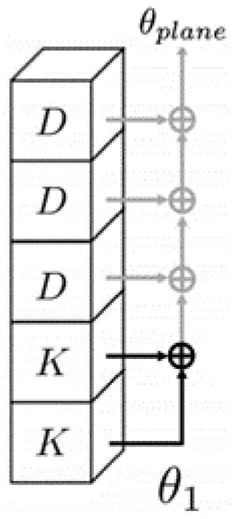
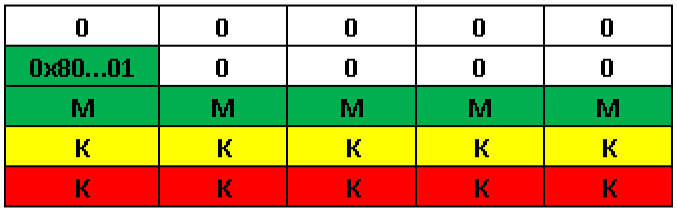


Figure 5.38: MAC-Keccak selection function target for two unknown key pieces [14].

Experiment 1 - Targeting bottom plane key

The objective of experiment 1 is to attack the current power model for targeting the θ_{plane} to extract the key at the bottom plane. In this experiment, it is assumed the 2nd key plane is part of the message even though, in reality, it is not. The purpose of this is to see if using a plain with static information will in any way affect the outcome of the CPA attack even though the single message plain has randomness that will impact the output θ_{plane} from the power model. From Figure 5.39, the target bytes to recover are in red, static information that is assumed to be part of the message is in yellow, and finally, the controlled messages are in green.

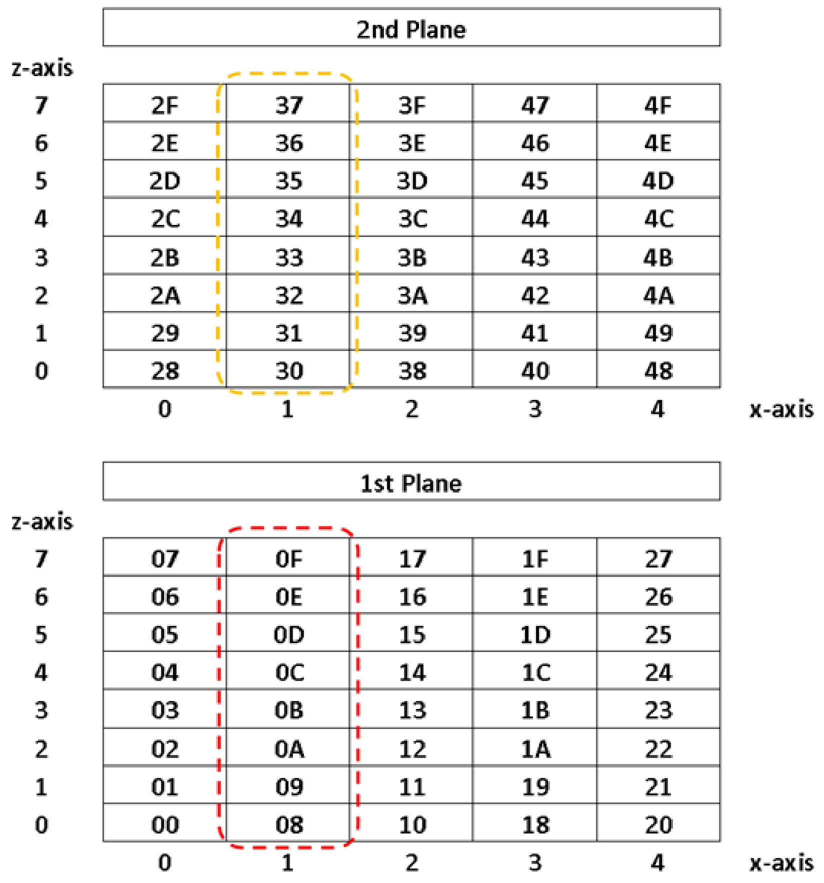


0	0	0	0	0
0x80...01	0	0	0	0
M	M	M	M	M
K	K	K	K	K
K	K	K	K	K

Figure 5.39: Assuming the 2nd key plane is part of the message and attempting to recover the 1st key plane.

Another way to view the target key bytes (or key lane to be specific) for this experiment is by looking at the top view of the Keccak state for the 1st and 2nd planes depicted in

Figure 5.40 where the target key lane has a red outline and the static lane information as part of the message has a yellow outline. Basically, the target is the 2^{nd} key lane of the bottom plane.



2nd Plane

z-axis					
7	2F	37	3F	47	4F
6	2E	36	3E	46	4E
5	2D	35	3D	45	4D
4	2C	34	3C	44	4C
3	2B	33	3B	43	4B
2	2A	32	3A	42	4A
1	29	31	39	41	49
0	28	30	38	40	48
	0	1	2	3	4	x-axis

1st Plane

z-axis					
7	07	0F	17	1F	27
6	06	0E	16	1E	26
5	05	0D	15	1D	25
4	04	0C	14	1C	24
3	03	0B	13	1B	23
2	02	0A	12	1A	22
1	01	09	11	19	21
0	00	08	10	18	20
	0	1	2	3	4	x-axis

Figure 5.40: Top view of the Keccak state, 1st and 2nd planes. Attacking the 1st plane key lane (red outline) by using the 2nd plane key lane (yellow outline) as part of the message. Each cell is a byte.

The choices for performing CPA attack are 4-bit and 8-bit CPA selection function key guess sizes with 16 and 256 key guesses, respectively. The reason for doing this is to see if the 4-bit CPA selection function key guess attack provides sufficient information that could be used in a 4-bit x 4-bit cascade CPA attack. If the information, indeed, could be used in a 4-bit x 4-bit cascade CPA, there is no reason to use a 8-bit x 8-bit cascade CPA as that is more computational intensive due to the number of available key guesses compared to the same number of key guesses 4-bit x 4-bit cascade CPA has to go through. The experiment uses 200K traces and 4-bit and 8-bit CPA selection function key guess size attacks. Figure

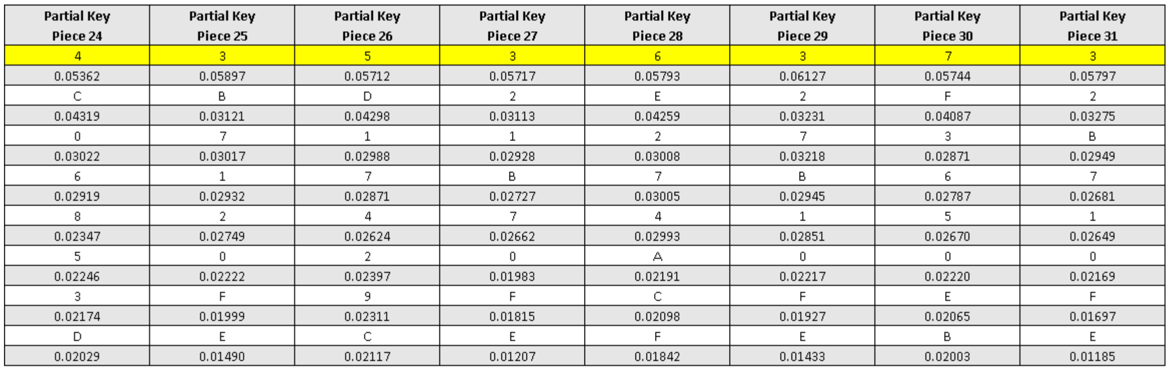
5.41 shows the result of the ranked top-8 correlation values of partial key pieces of Key1 in lane 1 from 8-bit CPA selection function key guess size attacks. Likewise, the same information is shown in Figure 5.42 and Figure 5.43 from 4-bit CPA selection function key guess size attacks. Observing the results from 4-bit and 8-bit CPA selection function key guess size attacks, the target key byte (1st plane key) has the value of the assumed key message which is why cells are color coded yellow to match with the assumed key message values as seen in the Keccak top view of Figure 5.40. Notice the different labeling of the 'Partial Key Piece' between 8-bit and 4-bit attacks. Since the target key lane is lane 1 of the bottom plane and each lane has 64 bits, for 8-bit CPA attack the target partial key piece starts at 8 (indexed 0) and goes up to 15. Similar labeling is used for the target key piece for 4-bit CPA. Therefore, the starting target key piece for lane 1 with 4-bit CPA is 16 and it goes up to 31. Even though 8-bit and 4-bit CPA selection function key guess size attacks provide the same key guess results, the correlation values are different for the two attacks. It makes sense for the 8-bit CPA selection function key guess size attack to have higher correlation values due to the better SNR as a result of the number of processed and available key guesses. It makes sense for the target key byte (1st plane key) to have the value of the assumed key message because of the XOR operation of the θ_{plane} operation. The key values on plane 1 and 2 canceled one another, i.e. where identical to each other. From this cancellation between the two keys, there is more degree-of-freedom to utilize the randomness in the message plane to influence the calculation of the correlation values for the power model, hence the attack of the bottom key lane turns out to be the values of the top key lane that is assumed to be part of the message.

Partial Key Piece 8	Partial Key Piece 9	Partial Key Piece 10	Partial Key Piece 11	Partial Key Piece 12	Partial Key Piece 13	Partial Key Piece 14	Partial Key Piece 15
30	31	32	33	34	35	36	37
0.08014	0.08315	0.08146	0.08350	0.07972	0.08085	0.08429	0.08152
10	30	3A	3B	30	25	37	27
0.06251	0.06365	0.06477	0.06483	0.06313	0.06254	0.06464	0.06380
38	11	36	31	36	31	32	3F
0.06180	0.06291	0.06400	0.06398	0.06244	0.06152	0.06461	0.06339
31	35	72	32	B4	3D	34	B7
0.06023	0.06286	0.06152	0.06289	0.06012	0.06134	0.06445	0.06146
32	33	B2	37	3C	15	26	33
0.06000	0.06253	0.06144	0.06273	0.05977	0.06111	0.06379	0.06117
B0	39	33	B3	74	37	76	36
0.05967	0.06247	0.06097	0.06231	0.05929	0.06069	0.06364	0.06067
34	71	22	73	14	B5	B6	35
0.05934	0.06183	0.06018	0.06197	0.05865	0.05965	0.06176	0.05977
20	21	30	13	35	75	3E	77
0.05877	0.06160	0.05848	0.06155	0.05759	0.05915	0.06161	0.05950

Figure 5.41: Top-8 correlation values of partial key pieces of Key1 in lane 1 of the first plane from 8-bit CPA selection function key guess size. Partial key pieces are in bytes and represented as hex values.

Partial Key Piece 16	Partial Key Piece 17	Partial Key Piece 18	Partial Key Piece 19	Partial Key Piece 20	Partial Key Piece 21	Partial Key Piece 22	Partial Key Piece 23
0	3	1	3	2	3	3	3
0.05595	0.05739	0.05748	0.06017	0.05495	0.06026	0.05632	0.06177
8	1	9	1	A	7	B	B
0.03955	0.03249	0.04500	0.03148	0.04361	0.03199	0.04352	0.03174
1	B	0	7	6	B	1	7
0.02776	0.02833	0.02967	0.03008	0.03025	0.03193	0.02882	0.03140
2	2	5	2	3	2	7	1
0.02739	0.02726	0.02870	0.02962	0.02600	0.03023	0.02703	0.03081
4	7	3	B	8	1	2	2
0.02673	0.02661	0.02838	0.02951	0.02487	0.02645	0.02690	0.02967
7	0	8	0	E	F	F	F
0.02399	0.02633	0.02370	0.02200	0.02267	0.01992	0.02272	0.02030
9	F	B	F	0	0	A	0
0.02205	0.02004	0.02212	0.01950	0.02225	0.01849	0.02099	0.01824
A	D	D	D	5	E	4	E
0.02073	0.01299	0.02158	0.01364	0.01828	0.01709	0.01984	0.01234

Figure 5.42: Top-8 correlation values of partial key pieces of Key1 in lane 1 of the first plane from 4-bit CPA selection function key guess size, targeting the first 4 key bytes of lane 1. Partial key pieces are in bytes and represented as hex values.

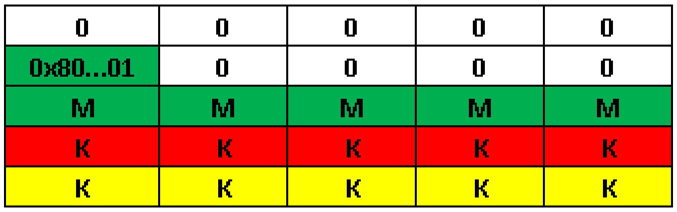


Partial Key Piece 24	Partial Key Piece 25	Partial Key Piece 26	Partial Key Piece 27	Partial Key Piece 28	Partial Key Piece 29	Partial Key Piece 30	Partial Key Piece 31
4	3	5	3	6	3	7	3
0.05362	0.05897	0.05712	0.05717	0.05793	0.06127	0.05744	0.05797
C	B	D	2	E	2	F	2
0.04319	0.03121	0.04298	0.03113	0.04259	0.03231	0.04087	0.03275
0	7	1	1	2	7	3	B
0.03022	0.03017	0.02988	0.02928	0.03008	0.03218	0.02871	0.02949
6	1	7	B	7	B	6	7
0.02919	0.02932	0.02871	0.02727	0.03005	0.02945	0.02787	0.02681
8	2	4	7	4	1	5	1
0.02347	0.02749	0.02624	0.02662	0.02993	0.02851	0.02670	0.02649
5	0	2	0	A	0	0	0
0.02246	0.02222	0.02397	0.01983	0.02191	0.02217	0.02220	0.02169
3	F	9	F	C	F	E	F
0.02174	0.01999	0.02311	0.01815	0.02098	0.01927	0.02065	0.01697
D	E	C	E	F	E	B	E
0.02029	0.01490	0.02117	0.01207	0.01842	0.01433	0.02003	0.01185

Figure 5.43: Top-8 correlation values of partial key pieces of Key1 in lane 1 of the first plane from 4-bit CPA selection function key guess size, targeting the last 4 key bytes of lane 1. Partial key pieces are in bytes and represented as hex values.

Experiment 2 - Targeting second plane key

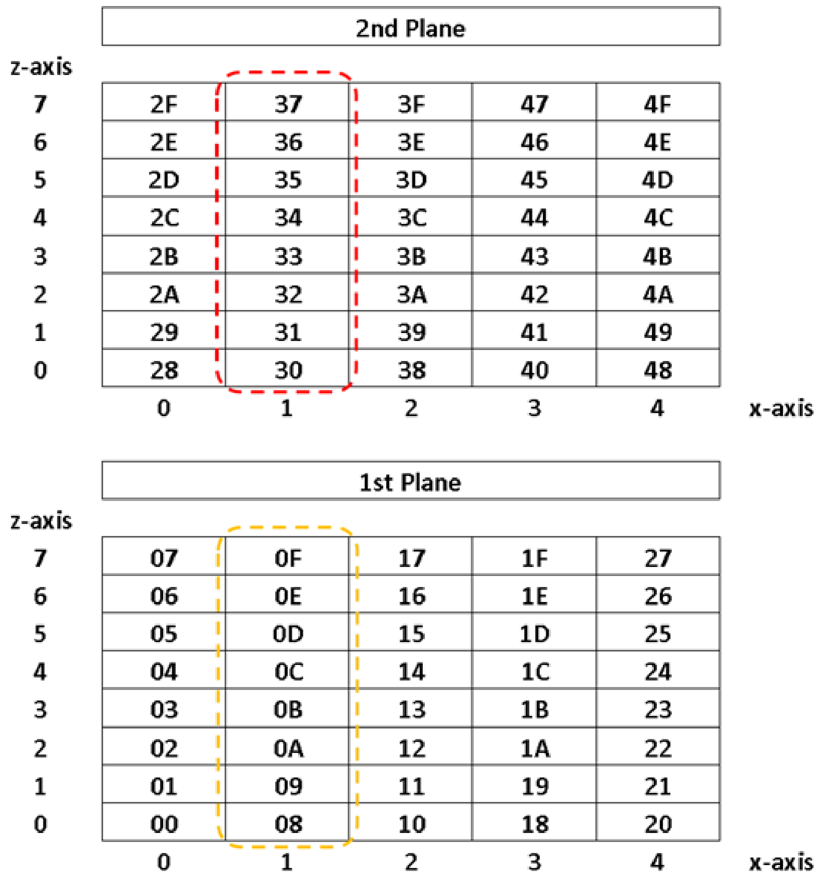
From the results of Experiment 1, the idea of targeting the second plane key leads one to conclude the results of the attack of lane 1 of the 2nd key plane will yield the key values from the bottom plane, which is indeed the case. The discussion for how this is happening or yielding the result of the key that is assumed to be part of the message can be referred back to the Experiment 1 analysis.



0	0	0	0	0
0x80...01	0	0	0	0
M	M	M	M	M
K	K	K	K	K
K	K	K	K	K

Figure 5.44: Assuming the 1st key plane is part of the message and attempting to recover the 2nd key plane.

From Figure 5.44, the target bytes to recover are in red, static information that is assumed to be part of the message is in yellow, and finally, the controlled messages are in green. Another way to view the target key bytes (or key lane, to be specific) for this experiment is by looking at the top view of the Keccak state for the 1st and 2nd planes depicted in Figure 5.45 where the target key lane has a red outline and the static lane information as part of the message has a yellow outline. Basically, the target is the 2nd key lane of the second plane.



2nd Plane

z-axis					
7	2F	37	3F	47	4F
6	2E	36	3E	46	4E
5	2D	35	3D	45	4D
4	2C	34	3C	44	4C
3	2B	33	3B	43	4B
2	2A	32	3A	42	4A
1	29	31	39	41	49
0	28	30	38	40	48
x-axis	0	1	2	3	4

1st Plane

z-axis					
7	07	0F	17	1F	27
6	06	0E	16	1E	26
5	05	0D	15	1D	25
4	04	0C	14	1C	24
3	03	0B	13	1B	23
2	02	0A	12	1A	22
1	01	09	11	19	21
0	00	08	10	18	20
x-axis	0	1	2	3	4

Figure 5.45: Top view of the Keccak state, 1st and 2nd planes. Attacking the 2nd plane key lane (red outline) by using the 1st plane key lane (yellow outline) as part of the message. Each cell is a byte.

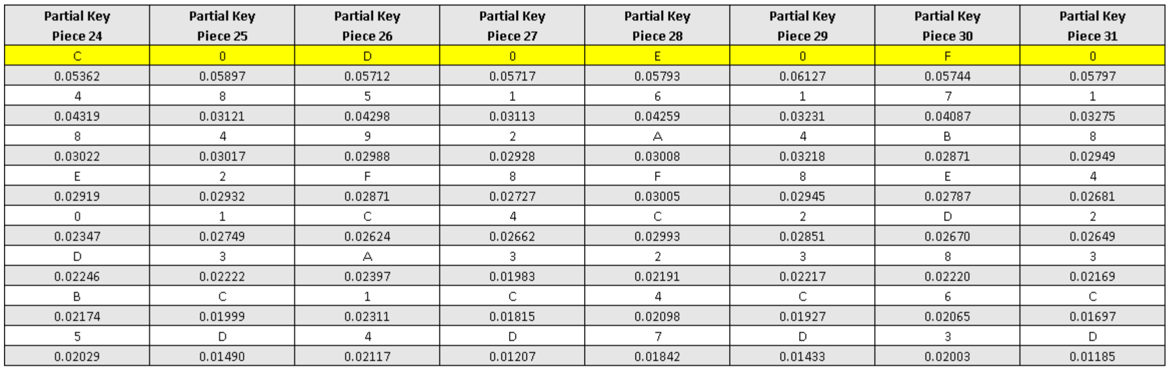
The choices for performing CPA attack are 4-bit and 8-bit CPA selection function key guess sizes with 16 and 256 key guesses, respectively. The experiment uses 200K traces and 4-bit and 8-bit CPA selection function key guess size attacks. Figure 5.46 shows the result of the ranked top-8 correlation values of partial key pieces of Key1 in lane 1 from 8-bit CPA selection function key guess size attacks. Likewise, Figure 5.47 and Figure 5.48 from 4-bit CPA selection function key guess size attacks. Observing the results from 4-bit and 8-bit CPA selection function key guess size attacks, the target key byte (2nd plane key) has the value of the assumed key message which is why cells are color-coded yellow to match with the assumed key message values as seen in the Keccak top view of Figure 5.45. As discussed in Experiment 1, the reason this is happening is the cancellation between the two keys that creates more degrees-of-freedom to utilize the randomness in the message plain to affect the θ_{plane} output that influences the calculation of the correlation values from the power model.

Partial Key Piece 8	Partial Key Piece 9	Partial Key Piece 10	Partial Key Piece 11	Partial Key Piece 12	Partial Key Piece 13	Partial Key Piece 14	Partial Key Piece 15
08	09	0A	0B	0C	0D	0E	0F
0.08014	0.08315	0.08146	0.08350	0.07972	0.08085	0.08429	0.08152
28	08	02	03	08	1D	0F	1F
0.06251	0.06365	0.06477	0.06483	0.06313	0.06254	0.06464	0.06380
00	29	0E	09	0E	09	0A	07
0.06180	0.06291	0.06400	0.06398	0.06244	0.06152	0.06461	0.06339
09	0D	4A	0A	8C	05	0C	8F
0.06023	0.06286	0.06152	0.06289	0.06012	0.06134	0.06445	0.06146
0A	0B	8A	0F	04	2D	1E	0B
0.06000	0.06253	0.06144	0.06273	0.05977	0.06111	0.06379	0.06117
88	01	0B	8B	4C	0F	4E	0E
0.05967	0.06247	0.06097	0.06231	0.05929	0.06069	0.06364	0.06067
0C	49	1A	4B	2C	8D	8E	0D
0.05934	0.06183	0.06018	0.06197	0.05865	0.05965	0.06176	0.05977
18	19	08	2B	0D	4D	06	4F
0.05877	0.06160	0.05848	0.06155	0.05759	0.05915	0.06161	0.05950

Figure 5.46: Top-8 correlation values of partial key pieces of Key1 in lane 1 of the second plane from 8-bit CPA selection function key guess size. Partial key pieces are in bytes and represented as hex values.

Partial Key Piece 16	Partial Key Piece 17	Partial Key Piece 18	Partial Key Piece 19	Partial Key Piece 20	Partial Key Piece 21	Partial Key Piece 22	Partial Key Piece 23
8	0	9	0	A	0	B	0
0.05595	0.05739	0.05748	0.06017	0.05495	0.06026	0.05632	0.06177
0	2	1	2	2	4	3	8
0.03955	0.03249	0.04500	0.03148	0.04361	0.03199	0.04352	0.03174
9	8	8	4	E	8	9	4
0.02776	0.02833	0.02967	0.03008	0.03025	0.03193	0.02882	0.03140
A	1	D	1	B	1	F	2
0.02739	0.02726	0.02870	0.02962	0.02600	0.03023	0.02703	0.03081
C	4	B	8	0	2	A	1
0.02673	0.02661	0.02838	0.02951	0.02487	0.02645	0.02690	0.02967
F	3	0	3	6	C	7	C
0.02399	0.02633	0.02370	0.02200	0.02267	0.01992	0.02272	0.02030
1	C	3	C	8	3	2	3
0.02205	0.02004	0.02212	0.01950	0.02225	0.01849	0.02099	0.01824
2	E	5	E	D	D	C	D
0.02073	0.01299	0.02158	0.01364	0.01828	0.01709	0.01984	0.01234

Figure 5.47: Top-8 correlation values of partial key pieces of Key1 in lane 1 of the second plane from 4-bit CPA selection function key guess size, targeting the first 4 key bytes of lane 1. Partial key pieces are in bytes and represented as hex values.



Partial Key Piece 24	Partial Key Piece 25	Partial Key Piece 26	Partial Key Piece 27	Partial Key Piece 28	Partial Key Piece 29	Partial Key Piece 30	Partial Key Piece 31
C	0	D	0	E	0	F	0
0.05362	0.05897	0.05712	0.05717	0.05793	0.06127	0.05744	0.05797
4	8	5	1	6	1	7	1
0.04319	0.03121	0.04298	0.03113	0.04259	0.03231	0.04087	0.03275
8	4	9	2	A	4	B	8
0.03022	0.03017	0.02988	0.02928	0.03008	0.03218	0.02871	0.02949
E	2	F	8	F	8	E	4
0.02919	0.02932	0.02871	0.02727	0.03005	0.02945	0.02787	0.02681
0	1	C	4	C	2	D	2
0.02347	0.02749	0.02624	0.02662	0.02993	0.02851	0.02670	0.02649
D	3	A	3	2	3	8	3
0.02246	0.02222	0.02397	0.01983	0.02191	0.02217	0.02220	0.02169
B	C	1	C	4	C	6	C
0.02174	0.01999	0.02311	0.01815	0.02098	0.01927	0.02065	0.01697
5	D	4	D	7	D	3	D
0.02029	0.01490	0.02117	0.01207	0.01842	0.01433	0.02003	0.01185

Figure 5.48: Top-8 correlation values of partial key pieces of Key1 in lane 1 of the second plane from 4-bit CPA selection function key guess size, targeting the last 4 key bytes of lane 1. Partial key pieces are in bytes and represented as hex values.

Experiment 3 - n-bit x n-bit cascade CPA attack

The results of Experiment 1 and 2, whether using 4-bit or 8-bit CPA selection function key guess size attacks, look promising as these experiments have narrowed down 2 correct key pieces out of 16 for 4-bit CPA and out of 256 for 8-bit CPA selection function key guess size attacks. However, the caveat is that it is uncertain the order of the key pieces. With the attack result characteristics of CPA using the power model to target θ_{plane} from Experiment 1 and 2, one would assume applying n-bit x n-bit cascade CPA attack can recover the 2 key pieces (4-bit or 8-bit wide) and is much better than knowing the parity of the two known key pieces in the same column using the regular n-bit CPA selection function key guess size attack. When applying 4-bit x 4-bit (256 guesses) cascade CPA with power model targeting θ_{plane} operation attack, the result is meaningless correlation values. Likewise, when applying 8-bit x 8-bit (65536) cascade CPA with power model targeting θ_{plane} operation attack, the result is also meaningless correlation values. Therefore, n-bit x n-bit cascade CPA with power model targeting θ_{plane} operation attack is not sufficient to attack 2 key planes.

Chapter 6

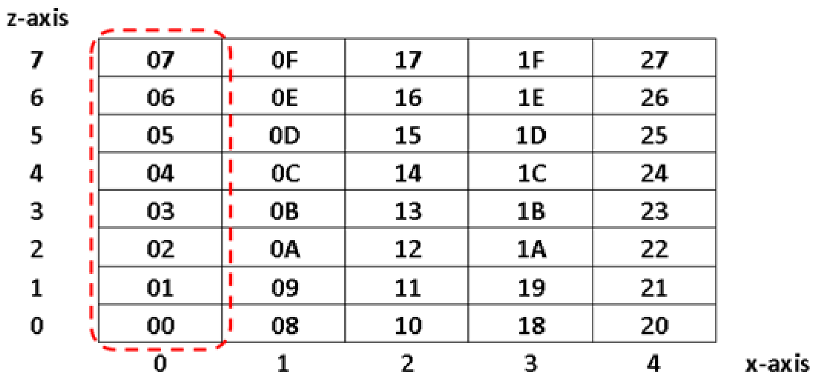
Results and Analysis

In the Algorithm Design section of the previous chapter, it discusses and shows that an attacker could exploit MAC-Keccak to retrieve a single key byte of a 40-byte long key by performing a power analysis attack on the θ_{plane} of the first Keccak round θ transformation. The CPA attack uses both 24 rounds and first 2 rounds of Keccak with 200K traces to retrieve the target key byte successfully. From that section, it also discusses any key length larger than 40 bytes is safe from CPA attacks on the θ_{plane} of the first Keccak round. Therefore, this chapter will look at cases where the key length is less than or equal to 40 bytes and examine to see how many key bytes of the 40-byte long of two different keys (one has internal byte values increased sequentially and another is from random generation) are able to retrieve from using the same CPA attack and power model. In addition, there will be analysis of the 1-bit, 2-bit, 4-bit, 8-bit, and 16-bit CPA selection function key guess size attacks that are performed on the waveforms to compare/analyze the optimization and computation effort/performance of successful key extraction on MAC-Keccak between using 24 rounds versus 2 rounds of Keccak. Please note the index labeling of any part of the Keccak state array such as the first lane is labeled 0 as the dimension is indexed by 0. The results were obtained from running the attacks on a Windows 7 PC using Intel Core i5, 32GB of RAM and 1TB hard drive.

Key1 - Attacked 1 Key Lane (8 Bytes)

By successfully attacking 1 key byte, the next step is to attack a single key lane that is 64 bits or 8 bytes as depicted in Figure 6.1 with encircled dotted red line. Figure 6.1 shows the key that is used in MAC-Keccak is Key1 and that it fits perfectly on the first plane of the

Keccak state. Key1 is defined as 40 bytes long and the value of the key starts from 0x00 to 0x27 sequentially.



z-axis					
7	07	0F	17	1F	27
6	06	0E	16	1E	26
5	05	0D	15	1D	25
4	04	0C	14	1C	24
3	03	0B	13	1B	23
2	02	0A	12	1A	22
1	01	09	11	19	21
0	00	08	10	18	20
	0	1	2	3	4
					x-axis

Figure 6.1: Top view of the Keccak state, attack target of 1 key lane of Key1 on the 1st plane. Each cell is a byte.

The following figures show the success rate of attacking 8 key bytes with different n-bit CPA selection function key guess size attacks.

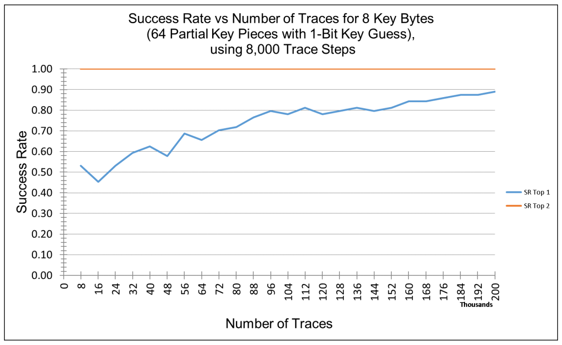


Figure 6.2: Success rates of 1-bit CPA selection function key guess size attacking 8 key bytes using 24 rounds of Keccak.

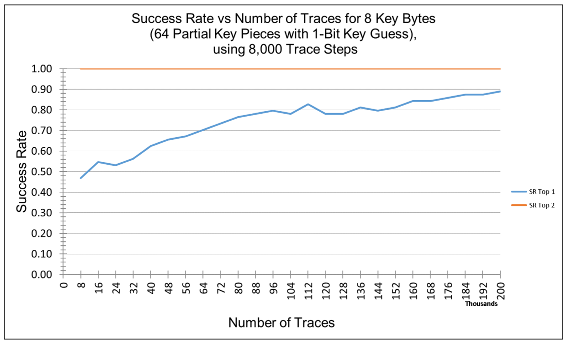


Figure 6.3: Success rates of 1-bit CPA selection function key guess size attacking 8 key bytes using 2 rounds of Keccak.

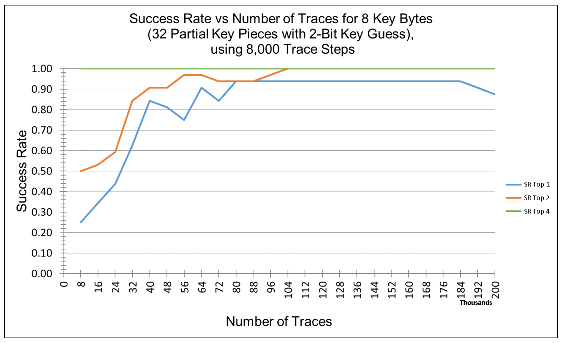


Figure 6.4: Success rates of 2-bit CPA selection function key guess size attacking 8 key bytes using 24 rounds of Keccak.

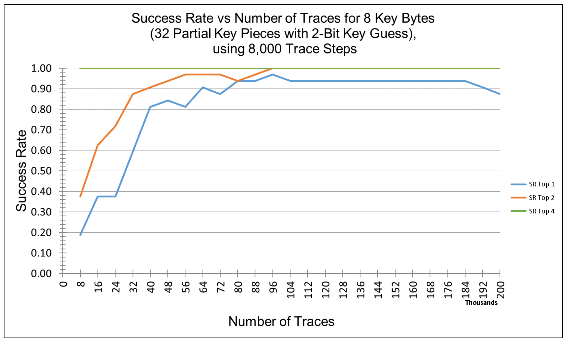


Figure 6.5: Success rates of 2-bit CPA selection function key guess size attacking 8 key bytes using 2 rounds of Keccak.

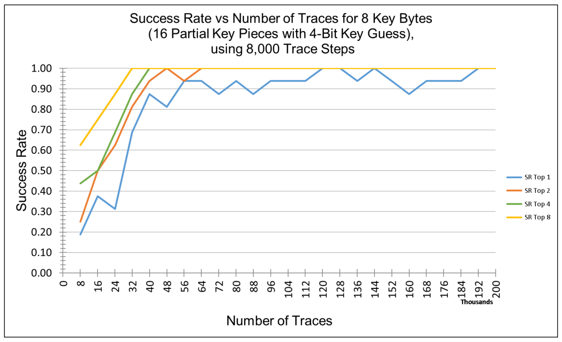


Figure 6.6: Success rates of 4-bit CPA selection function key guess size attacking 8 key bytes using 24 rounds of Keccak.

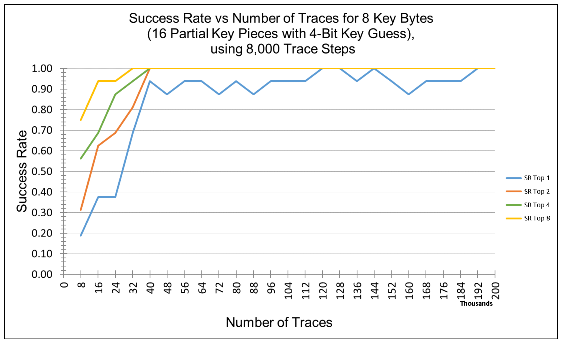


Figure 6.7: Success rates of 4-bit CPA selection function key guess size attacking 8 key bytes using 2 rounds of Keccak.

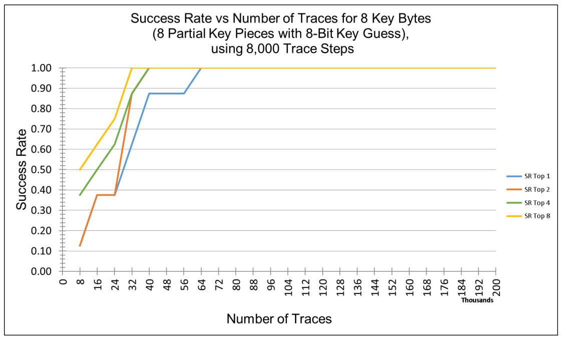


Figure 6.8: Success rates of 8-bit CPA selection function key guess size attacking 8 key bytes using 24 rounds of Keccak.

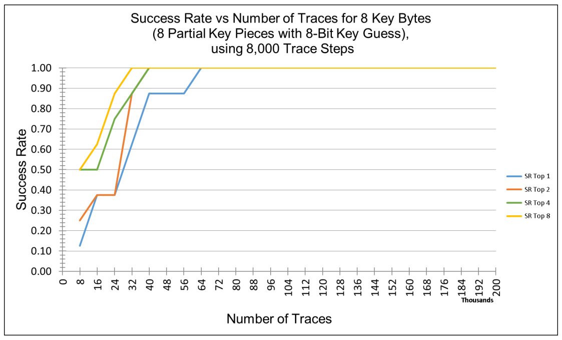


Figure 6.9: Success rates of 8-bit CPA selection function key guess size attacking 8 key bytes using 2 rounds of Keccak.

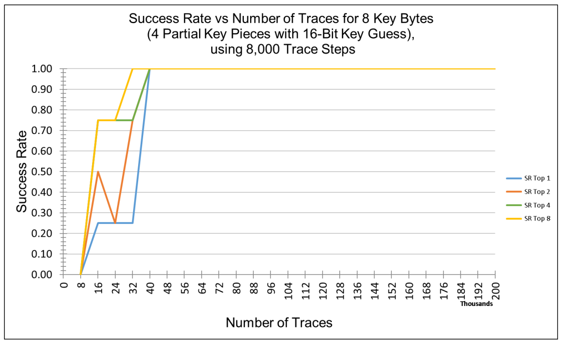


Figure 6.10: Success rates of 16-bit CPA selection function key guess size attacking 8 key bytes using 24 rounds of Keccak.

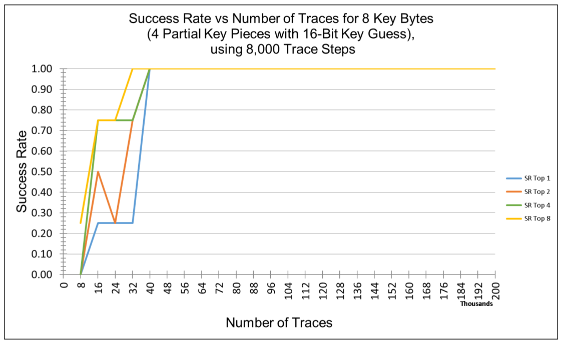


Figure 6.11: Success rates of 16-bit CPA selection function key guess size attacking 8 key bytes using 2 rounds of Keccak.

Figure 6.2 to Figure 6.11 show the success rates of n-bit CPA selection function key guess size for using 24 rounds and 2 rounds of Keccak. The figures on the left side result from using 24 rounds of Keccak from the power waveforms. Likewise, the figures on the right side show the data from only using/focusing on the first 2 rounds of Keccak from a Keccak that was simulated for 24 rounds. The legend of ‘SR top N’ indicates the target key piece is in the top N ranked correlation results. For example, ‘SR top 8’ indicates the value of the target key piece is in the top 8 correlation results, assuming the CPA attack application has access to the key and compares with the top 8 correlation results. Obviously, in real life, the CPA attack application does not have access to the key in order to compute the success rates. The goal here is to see how MAC-Keccak is susceptible to power analysis attacks from the research perspective. Comparing the results between the figures on the left using 24 rounds of Keccak versus the figures on the right using 2 rounds of Keccak, only the first few thousand applied traces are where using 24 rounds differs from using 2 rounds. This is more noticeable with using 1-bit, 2-bit and 4-bit CPA selection function key guess sizes up to 54K traces. With 8-bit and 16-bit CPA selection function key guess sizes, it is only noticeable up to 16K traces between using 24 rounds and 2 rounds of Keccak. The cause for this difference in the success rates is that when using 24 round Keccak, the correlation computation with small number traces between the 3rd and 24th rounds provide false positive correlation results for the correct key when, in fact, there should only be correlated values at the first round of Keccak. As more traces are applied to the attack, the correlation result of the correct key bytes standout more with the Keccak first round, and the success rates for n-bit CPA selection function key guess sizes are the same for using 24 rounds and 2 rounds. Another characteristic of these figures is that as the success rates approach 1, the threshold of the correlation result to contain the correct key value is relaxed such that SR top 8 approaches 1 faster than SR top 2. Only SR top 1 of n-bit CPA selection function key guess sizes is meaningful to determine how successfully the CPA attack is able to retrieve the key. Therefore, from these figures, as n-bit CPA selection function key guess size increases, SR top 1 approaches the value 1 with a fewer number of traces. It makes sense that 16-bit CPA selection function key guess size requires a fewer number of traces to retrieve the key byte successfully because there are 65,536 key guesses to correlate than compared to 4-bit CPA selection function key guess size with 16 key guesses to correlate. One can conclude that the larger the n-bit CPA selection function key guess size, the better the signal-to-noise-ratio (SNR), and that requires fewer numbers of traces needed to be applied to retrieve the key. Just to drive the point home, Figure 6.12 and Figure 6.13 show the usage of 24 rounds of Keccak and 2 rounds of Keccak, respectively, that larger n-bit CPA selection function key guess size with SR top 1 approaches the success rate of 1 with fewer number of traces and that as more traces are applied, the success rates from using two different Keccak rounds for CPA attacks are the same.

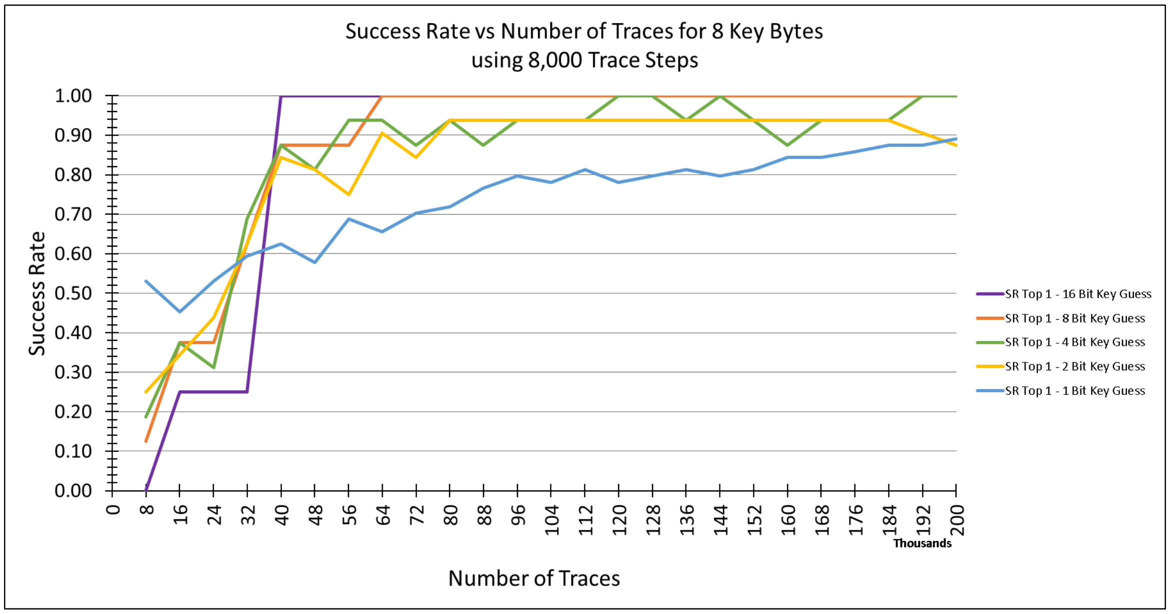


Figure 6.12: Success rates where the correct key is ranked first according to the correlation result for 1-bit, 2-bit, 4-bit, 8-bit and 16-bit CPA selection function key guess sizes when attacking 8 key bytes using 24 rounds of Keccak.

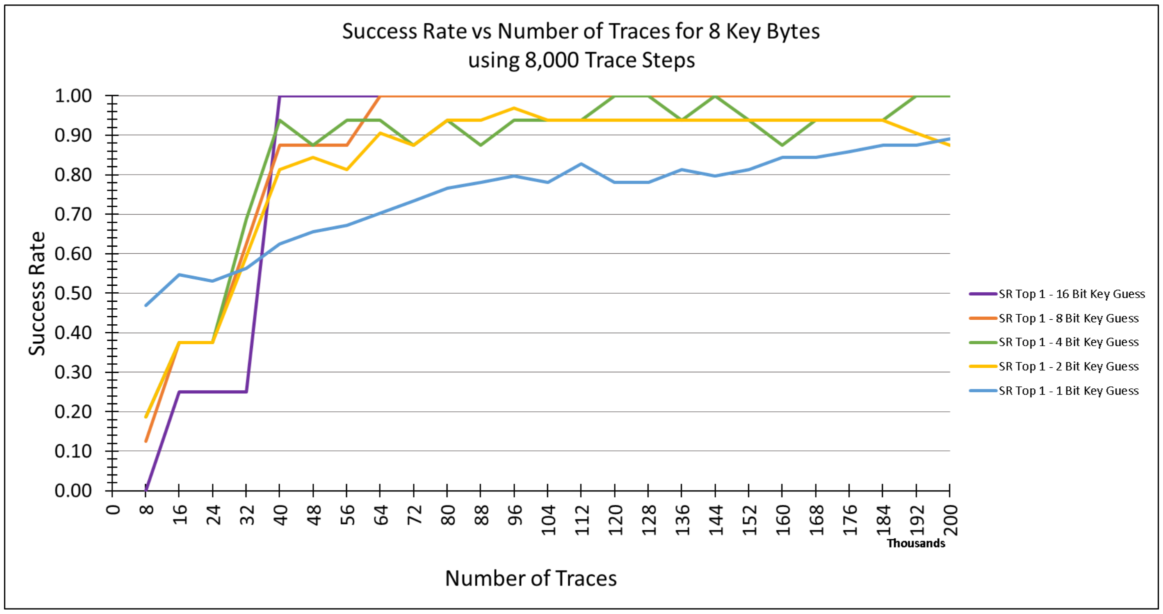


Figure 6.13: Success rates where the correct key is ranked first according to the correlation result for 1-bit, 2-bit, 4-bit, 8-bit and 16-bit CPA selection function key guess sizes when attacking 8 key bytes using 2 rounds of Keccak.

Figure 6.14 to Figure 6.17 show the runtime of n-bit CPA selection function key guess size for using 24 rounds and 2 rounds of Keccak. It is not surprising to see smaller n-bit CPA selection function key guess sizes such as the 4-bit CPA selection function key guess size having much better runtime than the 16-bit CPA selection function key guess size. Again, this is related to the number of key guesses for the chosen n-bit CPA selection function key guess size. The larger the n-bit CPA selection function key guess size, the more choices for the key guess and hence is more computation and runtime costly for CPA attacks. In addition to the size of the n-bit CPA selection function key guess size affecting the runtime, the number of applied traces is also another factor for the long runtime. The characteristic of the plots for the different n-bit CPA selection function key guess sizes is the same for the two different applied Keccak rounds. One takeaway point is using 2 rounds of Keccak is much faster in terms of the number of applied traces as discussed before and the runtime compared to using 24 rounds of Keccak due to the CPA algorithm processing much fewer power points. One can say that beyond the usage of the first 2 rounds of Keccak are extra power point information that do not add any more value than 2 rounds of Keccak for a successful attack of MAC-Keccak other than wasting computation resource and time.

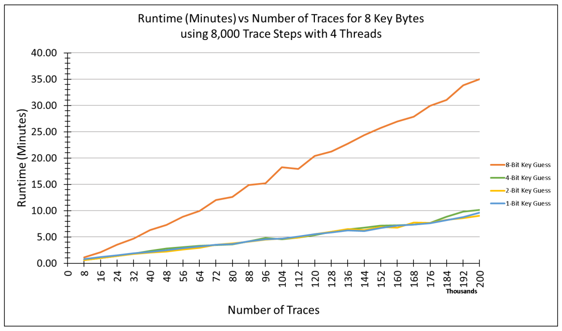


Figure 6.14: Runtime of 1-bit, 2-bit, 4-bit, and 8-bit CPA selection function key guess sizes using 24 rounds of Keccak.

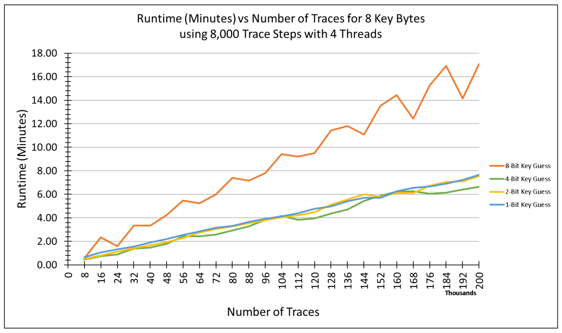


Figure 6.15: Runtime of 1-bit, 2-bit, 4-bit, and 8-bit CPA selection function key guess sizes using 2 rounds of Keccak.

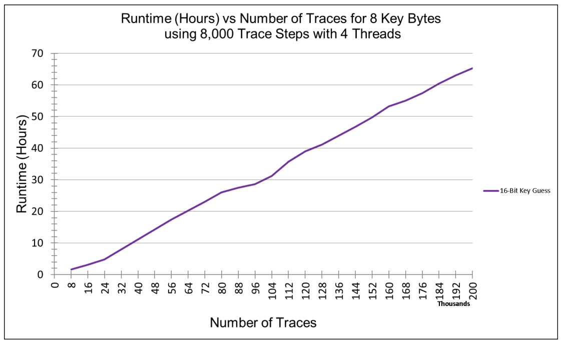


Figure 6.16: Runtime of 16-bit CPA selection function key guess size using 24 rounds of Keccak.

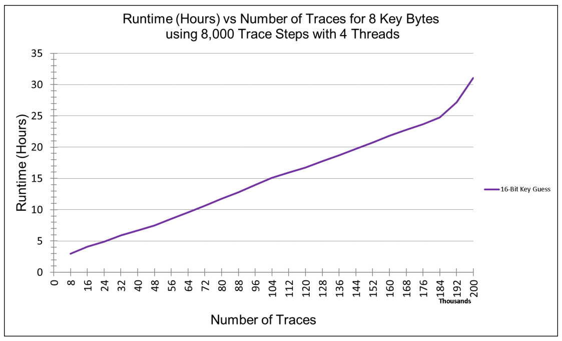


Figure 6.17: Runtime of 16-bit CPA selection function key guess size using 2 rounds of Keccak.

Figure 6.18 to Figure 6.21 show the cumulative runtime of n-bit CPA selection function key guess sizes using 24 rounds and 2 rounds of Keccak. The previous figures show the runtime of n-bit CPA selection function key guess sizes per applied traces, whereas the following figures show the cumulative runtime. The characteristic of the cumulative runtime curves is the same as the runtime curves with n-bit CPA selection function key guess size. The cumulative runtime is nothing more than the continuous time for the length of the CPA attacks using a fixed number of traces applied to each attack iteration. In this case, the attack of the 8 key bytes expands to 200K traces using 8K trace steps. It is interesting to note that the 16-bit CPA selection function key guess size requires fewer applied traces for successful attack of 8 key bytes, but looking at the cumulative runtime regardless of using 24 rounds or 2 rounds of Keccak, the computation time is far too long, i.e. number of days, compared to the 8-bit CPA selection function key guess size that requires only a few hours for 100% 8 byte key recovery.

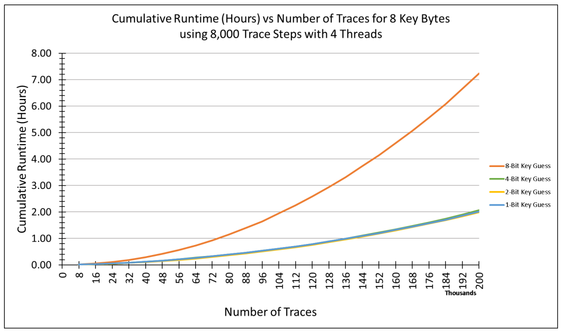


Figure 6.18: Cumulative runtime of 1-bit, 2-bit, 4-bit, and 8-bit CPA selection function key guess sizes using 24 rounds of Keccak.

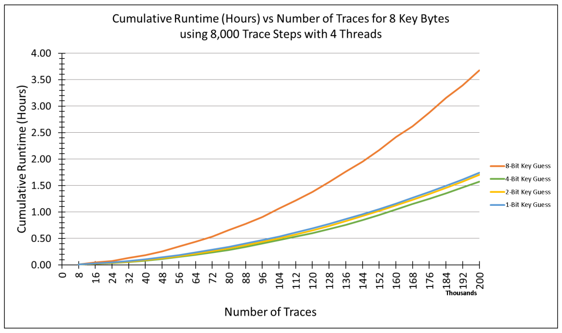


Figure 6.19: Cumulative runtime of 1-bit, 2-bit, 4-bit, and 8-bit CPA selection function key guess sizes using 2 rounds of Keccak.

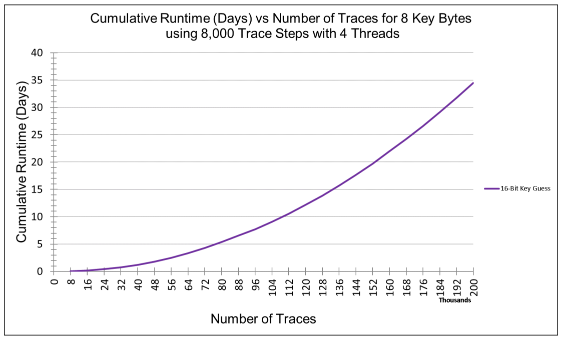


Figure 6.20: Cumulative runtime of 16-bit CPA selection function key guess size using 24 rounds of Keccak.

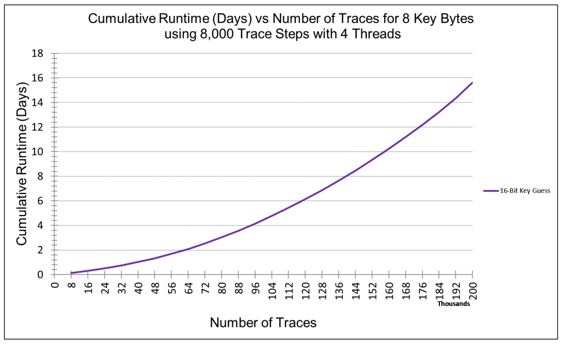


Figure 6.21: Cumulative runtime of 16-bit CPA selection function key guess size using 2 rounds of Keccak.

From analyzing the success rates, runtimes and cumulative runtimes of the n-bit CPA selection function key guess sizes, it is clear the 8-bit CPA selection function key guess size is the best candidate for CPA attack in terms of number of required traces and computation performance time to successfully attack 8 key bytes. The data also supports using 2 rounds of Keccak is optimal compared to 24 rounds of Keccak to provide the same correlation results as increasing the number of applied traces and benefits from much faster computation time.

Recall the following plot from the work of Pei *et al.* discussed in the Previous Work chapter. The plot shows the success rate for attacking a single key byte using physical traces from a SASEBO-GII board which contains a Xilinx Virtex-5 FPGA and also the same power model as this research that exploits the θ_{plane} of the first round Keccak θ transformation where the key is on the bottom plane before being diffused across the Keccak state from the other transformations. From the plot, applying 500K traces Pei *et al.* is able to recover one key byte with SR top 1 of around 90% using the 8-bit CPA selection function key guess size. This is understandable since these processed traces came from a physical device that is susceptible to measurement equipment and surrounding noise. Compared to the data result of the success rate using simulated traces depicted in Figure 6.13, usage of 8-bit and 16-bit CPA selection function key guess size attacks are able to retrieve not only 1 key byte but the whole key lane, i.e. 8 bytes, with a success rate of 1 using only 64K traces. What this means is that using simulated traces for studying power models to target Keccak operation is much more superior than using physical traces without the interface of noise such as environmental noise, measurement device noise, etc. Therefore, one can say using circuit simulation significantly reduces the complexity of mounting a power attack, provides quicker feedback during the implementation/study of a cryptographic device, and that ultimately reduces the cost of testing and experimentation.

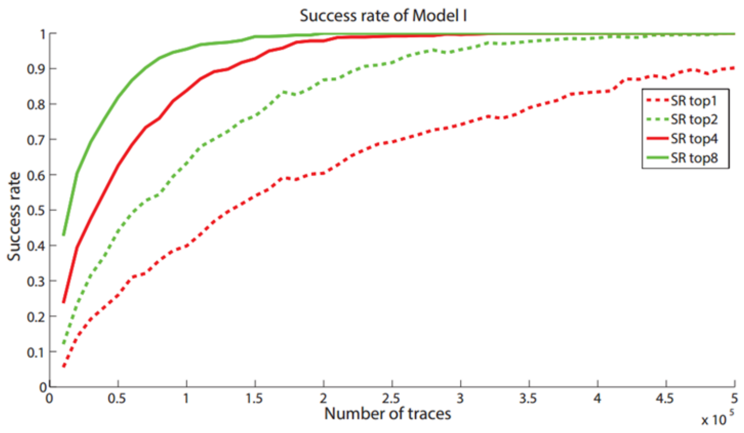


Figure 6.22: Success rate based on model 1 [15].

Key1 - Attacked 5 Key Lanes (40 Bytes)

Now that one is able to attack the first key lane of Key1 successfully, the next step is to attack all five key lanes which are on the bottom plane of the Keccak state. From the experiment results of the just discussed section where 8-bit CPA selection function key guess size shows to be the best candidate to use for CPA attacks in terms of the number of traces required for successful attacks and also providing the best computation runtime performance that is practical for real world attack. Figure 6.23 to Figure 6.28 show the results of attacking 40 key bytes of Key1 using 8-bit CPA selection function key guess size.

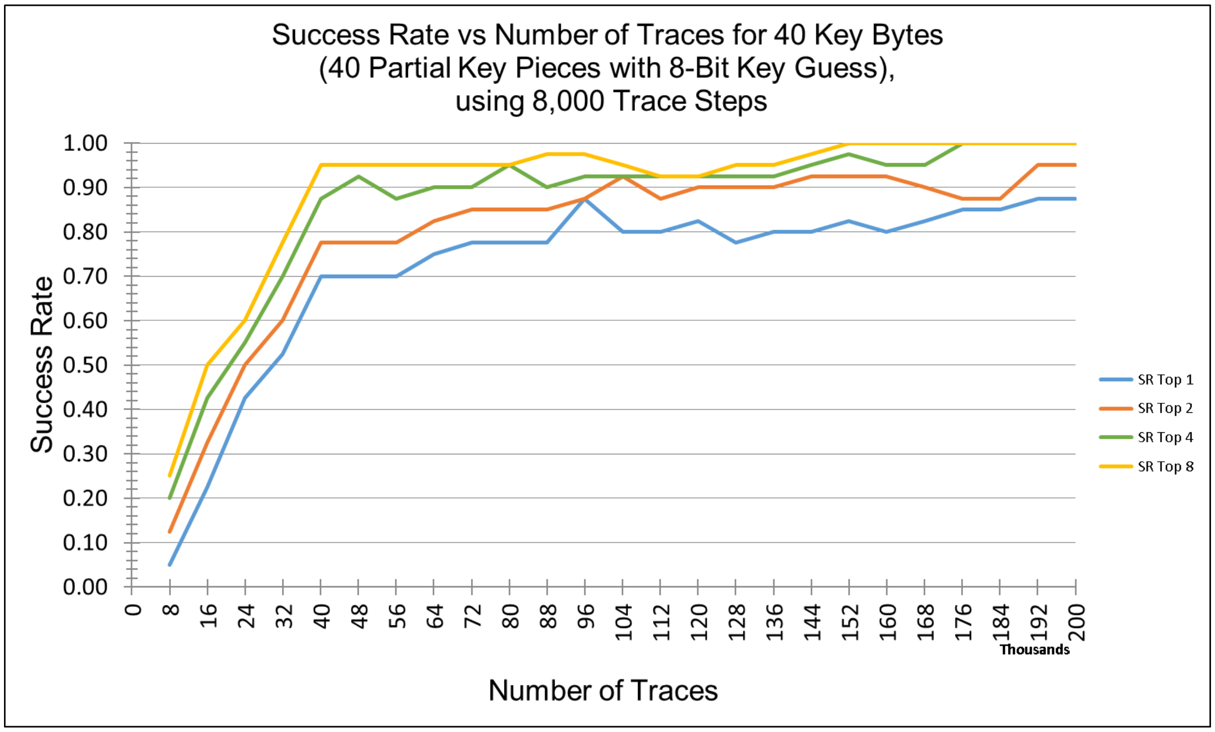


Figure 6.23: Success rates of 8-bit CPA selection function key guess size attacking 40 key bytes using 24 rounds of Keccak.

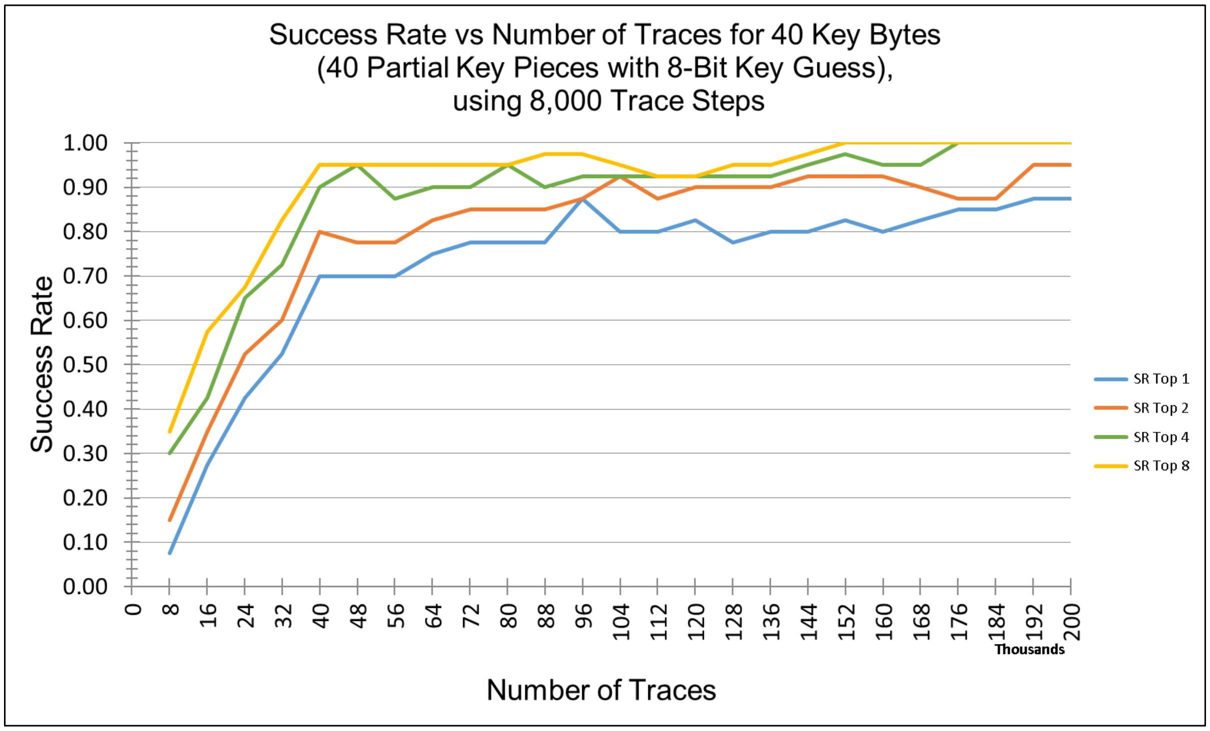


Figure 6.24: Success rates of 8-bit CPA selection function key guess size attacking 40 key bytes using 2 rounds of Keccak.

Figure 6.23 and Figure 6.24 show the success rate for attacking 40 key bytes with 8K trace steps using 24 rounds and 2 rounds of Keccak, respectively. The two figures look almost the same as applying more traces and confirm the usage of 2 rounds of Keccak is sufficient for a CPA attack targeting the θ_{plane} of the Keccak first round θ transformation. These SR top N curves for attacking 40 key bytes show the same characteristic as attacking a single key lane in the earlier mentioned figures.

The following four figures show the runtime and cumulative runtime of the 8-bit CPA selection function key guess size for 24 rounds and 2 rounds of Keccak, where the usage of the 2 rounds has a computation runtime twice as fast as using 24 rounds. These figures come in handy when developing and targeting new operations for power models as it may be useful to determine how long to execute CPA attacks with certain numbers of traces in order to test the new power model.

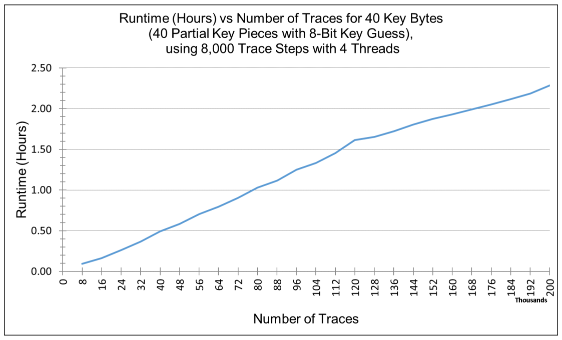


Figure 6.25: Runtime of 8-bit CPA selection function key guess size using 24 rounds of Keccak.

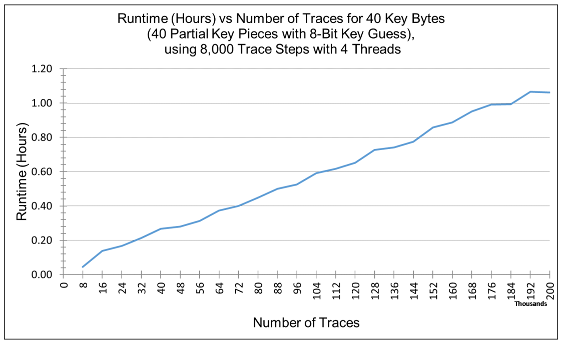


Figure 6.26: Runtime of 8-bit CPA selection function key guess size using 2 rounds of Keccak.

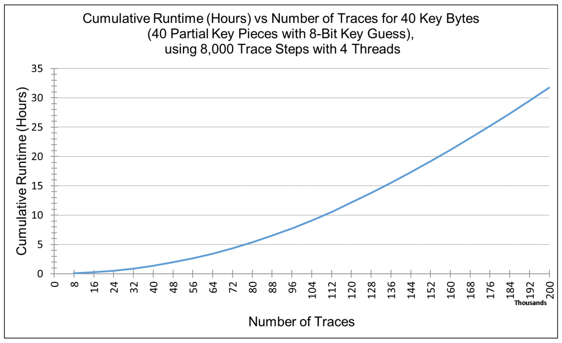


Figure 6.27: Cumulative runtime of 8-bit CPA selection function key guess size using 24 rounds of Keccak.

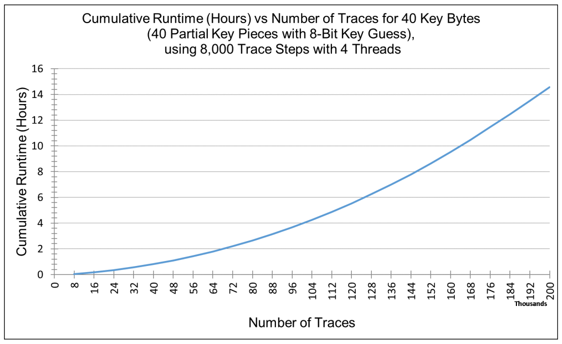
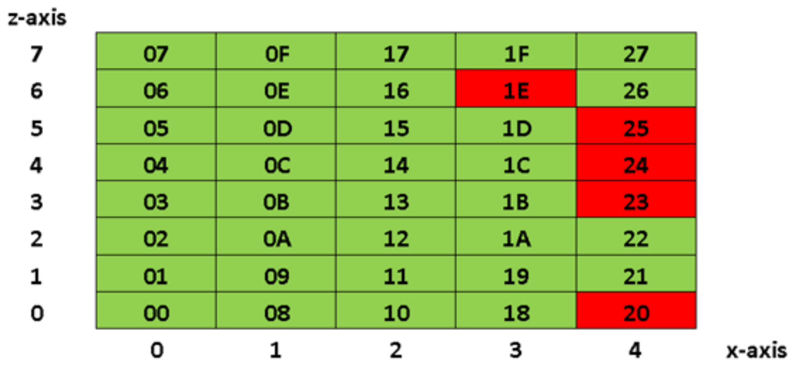


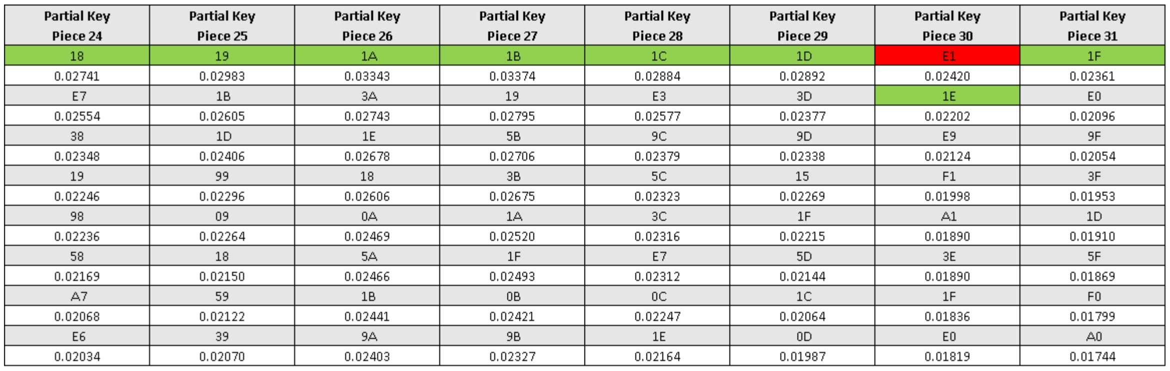
Figure 6.28: Cumulative runtime of 8-bit CPA selection function key guess size using 2 rounds of Keccak.

The summary of the 8-bit CPA selection function key guess size attack on Key1 for 40 bytes using 200K power traces is illustrated in Figure 6.29. The attack only recovers 35 of the 40 key bytes and that is an 87.5% success rate which can be confirmed from the SR top 1 of 8-bit CPA selection function key guess size of Figure 6.23 and Figure 6.24, where the highlighted green cells mean correct key recovery and red cells mean incorrect key recovery. Figure 6.30 shows the ranked top 8 correlation values of each target key byte of the 4th key lane and, similarly, Figure 6.31 shows the ranked top 8 correlation of each target key byte of the 5th key lane. By examining the top 8 correlation results of the incorrect guess key bytes that are depicted in Figure 6.30 and Figure 6.31, one can see that the correlation of the correct key guess for the targeted key byte is not that far off, at most 2 below the top correlation value. By appyling more traces, eventually the correct key guess will bubble to the top as the one with the highest correlation value, which means more intensive correlation computation between the power model and the power traces leading to longer runtime. It is interesting to see the incorrectly recovered target key bytes happen mostly on the last key lane. This is probably due to how the random messages are generated by the C code program, i.e. the messages tend to have the same bit sequence and not create enough randomness. Recall that the power model target θ_{plane} relies on the output of the simple linear XOR operation of the Keccak state column. If for some reason the bits in the column spit out the same value when using different messages to trigger the XOR, it will not be useful for the power model since it is not able to discern between using correct key guess and incorrect key guess. Therefore, for future work, one could generate input messages such that the bits of the column are aligned in a special way that will exploit the XOR operation in order to be considered as an effective power model.



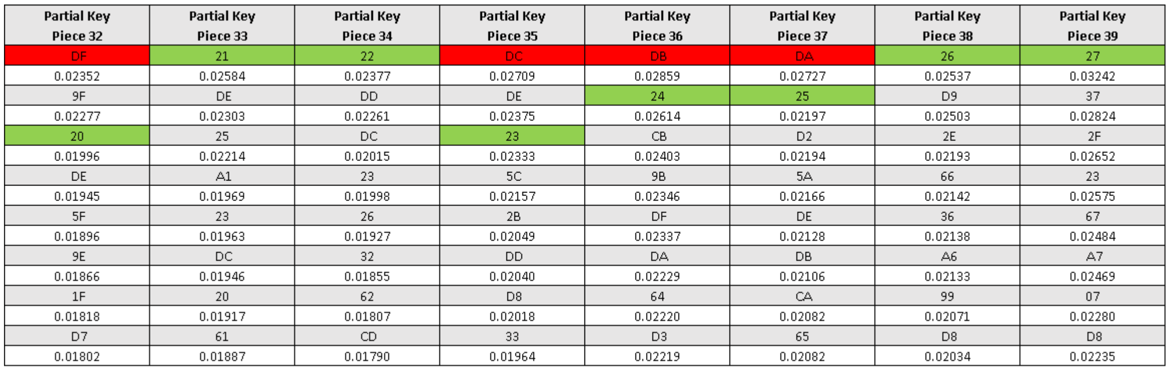
z-axis						
7	🟩 07	🟩 0F	🟩 17	🟩 1F	🟩 27	
6	🟩 06	🟩 0E	🟩 16	🟥 1E	🟩 26	
5	🟩 05	🟩 0D	🟩 15	🟩 1D	🟥 25	
4	🟩 04	🟩 0C	🟩 14	🟩 1C	🟥 24	
3	🟩 03	🟩 0B	🟩 13	🟩 1B	🟥 23	
2	🟩 02	🟩 0A	🟩 12	🟩 1A	🟩 22	
1	🟩 01	🟩 09	🟩 11	🟩 19	🟩 21	
0	🟩 00	🟩 08	🟩 10	🟩 18	🟥 20	
	0	1	2	3	4	x-axis

Figure 6.29: Recovered 35 of 40 key bytes of Key1 using 8-bit CPA selection function key guess size.



Partial Key Piece 24	Partial Key Piece 25	Partial Key Piece 26	Partial Key Piece 27	Partial Key Piece 28	Partial Key Piece 29	Partial Key Piece 30	Partial Key Piece 31
18	19	1A	1B	1C	1D	E1	1F
0.02741	0.02983	0.03343	0.03374	0.02884	0.02892	0.02420	0.02361
E7	1B	3A	19	E3	3D	1E	E0
0.02554	0.02605	0.02743	0.02795	0.02577	0.02377	0.02202	0.02096
38	1D	1E	5B	9C	9D	E9	9F
0.02348	0.02406	0.02678	0.02706	0.02379	0.02338	0.02124	0.02054
19	99	18	3B	5C	15	F1	3F
0.02246	0.02296	0.02606	0.02675	0.02323	0.02269	0.01998	0.01953
98	09	0A	1A	3C	1F	A1	1D
0.02236	0.02264	0.02469	0.02520	0.02316	0.02215	0.01890	0.01910
58	18	5A	1F	E7	5D	3E	5F
0.02169	0.02150	0.02466	0.02493	0.02312	0.02144	0.01890	0.01869
A7	59	1B	0B	0C	1C	1F	F0
0.02068	0.02122	0.02441	0.02421	0.02247	0.02064	0.01836	0.01799
E6	39	9A	9B	1E	0D	E0	A0
0.02034	0.02070	0.02403	0.02327	0.02164	0.01987	0.01819	0.01744

Figure 6.30: Top-8 correlation values of partial key pieces of Key1 in lane 3 of the bottom plane. Partial key pieces are in bytes and represented as hex values.

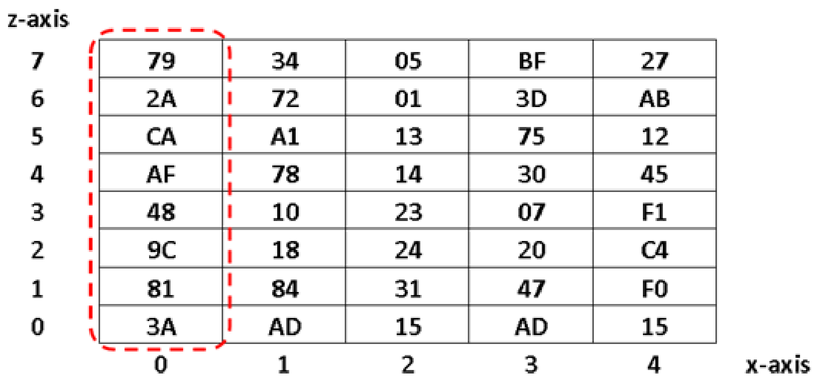


Partial Key Piece 32	Partial Key Piece 33	Partial Key Piece 34	Partial Key Piece 35	Partial Key Piece 36	Partial Key Piece 37	Partial Key Piece 38	Partial Key Piece 39
DF	21	22	DC	DB	DA	26	27
0.02352	0.02584	0.02377	0.02709	0.02859	0.02727	0.02537	0.03242
9F	DE	DD	DE	24	25	D9	37
0.02277	0.02303	0.02261	0.02375	0.02614	0.02197	0.02503	0.02824
20	25	DC	23	CB	D2	2E	2F
0.01996	0.02214	0.02015	0.02333	0.02403	0.02194	0.02193	0.02652
DE	A1	23	5C	9B	5A	66	23
0.01945	0.01969	0.01998	0.02157	0.02346	0.02166	0.02142	0.02575
5F	23	26	2B	DF	DE	36	67
0.01896	0.01963	0.01927	0.02049	0.02337	0.02128	0.02138	0.02484
9E	DC	32	DD	DA	DB	A6	A7
0.01866	0.01946	0.01855	0.02040	0.02229	0.02106	0.02133	0.02469
1F	20	62	D8	64	CA	99	07
0.01818	0.01917	0.01807	0.02018	0.02220	0.02082	0.02071	0.02280
D7	61	CD	33	D3	65	D8	D8
0.01802	0.01887	0.01790	0.01964	0.02219	0.02082	0.02034	0.02235

Figure 6.31: Top-8 correlation values of partial key pieces of Key1 in lane 4 of the bottom plane. Partial key pieces are in bytes and represented as hex values.

Key2 - Attacked 1 Key Lane (8 Bytes) Using 2 Rounds of Keccak

The research work then focuses on attacking another key that was generated randomly to confirm the CPA attack with the power model targeting the θ_{plane} of the θ transformation of the first round is really able to extract the key from MAC-Keccak successfully and that it was not a coincidence with Key1. Key1 was specifically chosen to have its 40 key bytes starting sequentially from 0x00 for the first byte to 0x27 for the last byte so it would be easy to monitor from the correlation results and also to debug the CPA attack algorithm. With the data results from retrieving Key1 and high confidence of the selected target operation for the power model for the CPA algorithm working correctly, Key2 is used to double check the attack algorithm for correctness. Figure 6.32 shows the layout of the 40 byte long Key2 across the bottom plane of the Keccak state and also the target key lane of 8 bytes for attack that is encircled with dotted red line.



z-axis					
7	79	34	05	BF	27
6	2A	72	01	3D	AB
5	CA	A1	13	75	12
4	AF	78	14	30	45
3	48	10	23	07	F1
2	9C	18	24	20	C4
1	81	84	31	47	F0
0	3A	AD	15	AD	15
	0	1	2	3	4 x-axis

Figure 6.32: Top view of the Keccak state, attack target of 1 key lane of Key2 on the 1st plane. Each cell is a byte.

Based on the data result from attacking Key1 with 24 rounds and 2 rounds of Keccak, the attacks on MAC-Keccak for Key2 are done with 2 rounds of Keccak to save computation time and memory. The following figures show the success rate of attacking 1 key lane also known as 8 key bytes with different n-bit CPA selection function key guess size attacks.

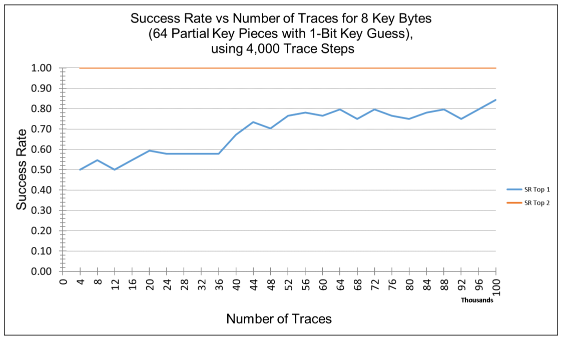


Figure 6.33: Success rates of 1-bit CPA selection function key guess size attacking 8 key bytes using 2 rounds of Keccak.

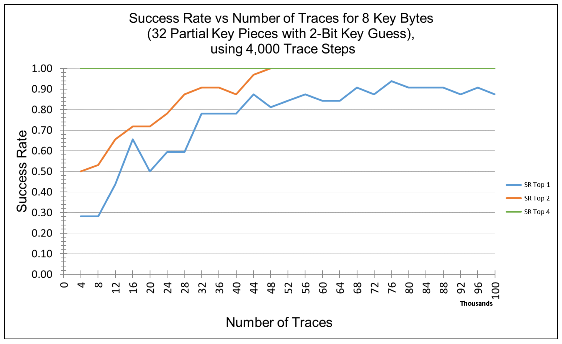


Figure 6.34: Success rates of 2-bit CPA selection function key guess size attacking 8 key bytes using 2 rounds of Keccak.

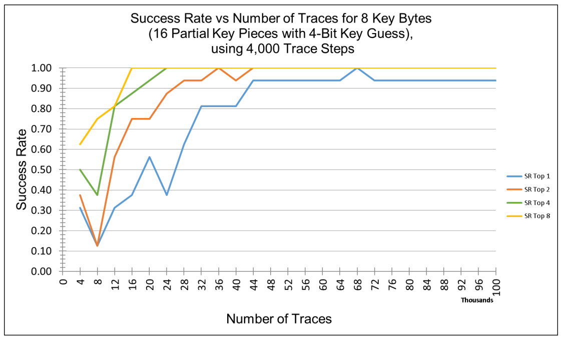


Figure 6.35: Success rates of 4-bit CPA selection function key guess size attacking 8 key bytes using 2 rounds of Keccak.

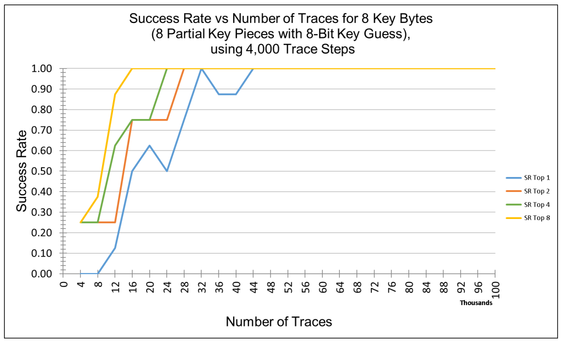


Figure 6.36: Success rates of 8-bit CPA selection function key guess size attacking 8 key bytes using 2 rounds of Keccak.

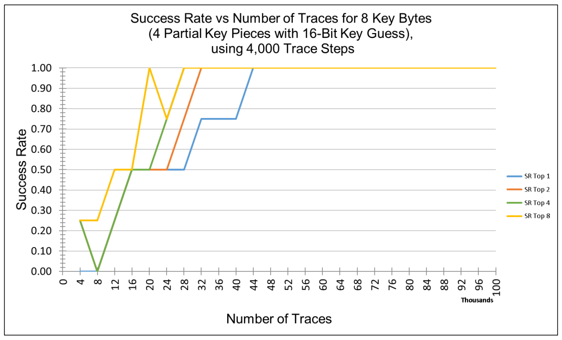


Figure 6.37: Success rates of 16-bit CPA selection function key guess size attacking 8 key bytes using 2 rounds of Keccak.

Based on the observation of the SR top 8 of the different n-bit CPA selection function key guess size attacks from Figure 6.33 to Figure 6.37, it confirms the finding from attacking Key1 that as the size of the CPA selection key guess size increases there will be better SNR and that requires fewer traces to use to successfully attack a whole key lane. Figure 6.38 shows only the SR top 1 of the different used n-bit CPA selection function key guess size attacks. It is interesting to see a 1-bit CPA selection function key guess size attack having around an 82% success rate of retrieving the whole key lane given the fact this CPA attack has only 2 possible key guesses. Another interesting point from this curve regarding attacking Key1 is that both 8-bit and 16-bit CPA selection function key guess size attacks approach the success rate of 1 with the same number of applied traces. In the earlier curve of SR top 1 of the different used n-bit CPA selection function key guess size attacks of a single key lane of Key1 in Figure 6.13 where the 16-bit CPA selection function key guess size clearly outperforms 8-bit CPA selection function key guess size to achieve a stable success rate of 1 in terms of requiring a fewer number of applied traces. Here with attacking a single lane of Key2, 8-bit and 16-bit CPA selection function key guess size attacks require 44K traces to achieve a stable success rate of 1. There are two things to be learned from this event. First, the data result confirms the 8-bit CPA selection function key guess size in some cases is as good as the 16 CPA selection function key guess size regarding the number of required traces to achieve a stable success rate of 1 given the fact the 8-bit CPA selection function key guess size has only 256 key guesses while the 16-bit CPA selection function key guess size has 65,536 key guesses which has better SNR. The second thing to note is that in the worst case scenario, the larger n-bit CPA selection function key guess size such as 16-bit cannot perform worse than the next lower n-bit CPA selection function key guess size which is 8-bit, in this case. Based on these two points, the success rates of n-bit CPA selection function key guess size attacks from Figure 6.38 show the higher the n-bit CPA selection function key guess size, the better the SNR and the fewer number of traces required for successful attacks. This statement can be backed up by observing the 1-bit, 2-bit and 4-bit selection function key guess size attacks from the plot that either the attacks never reach a success rate of 1 or if it does, it is not stable like with 8-bit and 16-bit selection function key guess size attacks.

Figure 6.39 to Figure 6.42 are other experiment results from attacking the first key lane of Key1 regarding the runtime and cumulative runtime of the different n-bit CPA selection function key guess sizes. The results from these curves further confirm that 8-bit CPA selection function key guess size is the best candidate for CPA attack, not only in terms of the required power traces to launch a successful attack, but also in terms of optimal runtime and cumulative runtime. 16-bit CPA selection function key guess size gives the best SNR but is not practical for real power analysis applications due to poor performance runtime and cumulative runtime.

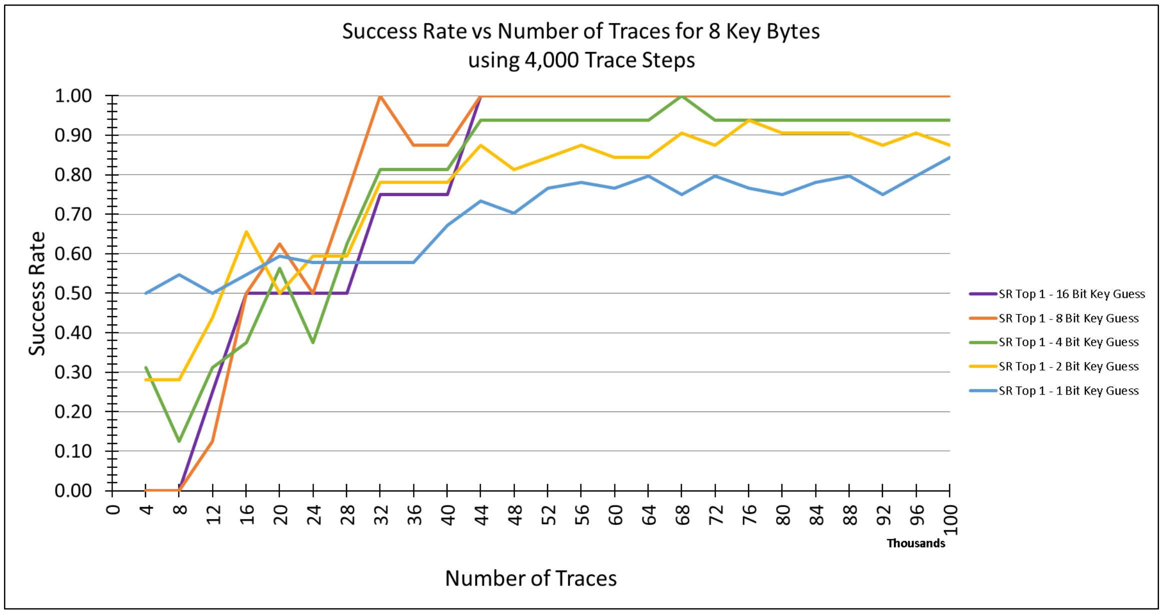


Figure 6.38: Success rates of the correct key that is ranked as top guess according to the correlation result for 1-bit, 2-bit, 4-bit, 8-bit and 16-bit CPA selection function key guess sizes attacking 8 key bytes using 2 rounds of Keccak.

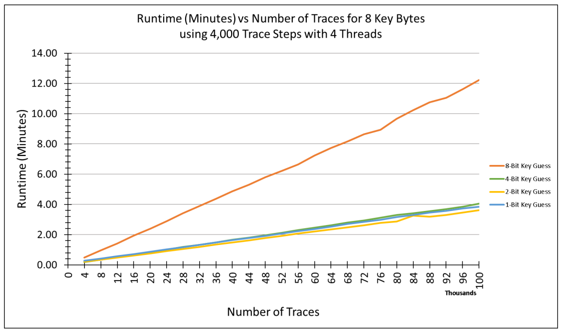


Figure 6.39: Runtime of 1-bit, 2-bit, 4-bit and 8-bit CPA selection function key guess sizes using 2 rounds of Keccak.

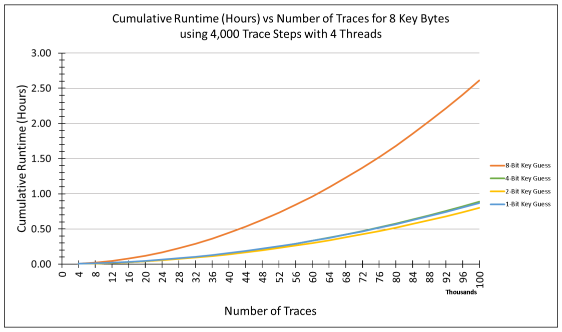


Figure 6.40: Cumulative runtime of 1-bit, 2-bit, 4-bit and 8-bit CPA selection function key guess sizes using 2 rounds of Keccak.

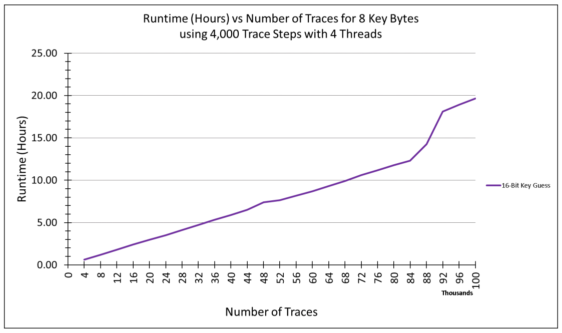


Figure 6.41: Runtime of 16-bit CPA selection function key guess size using 2 rounds of Keccak.

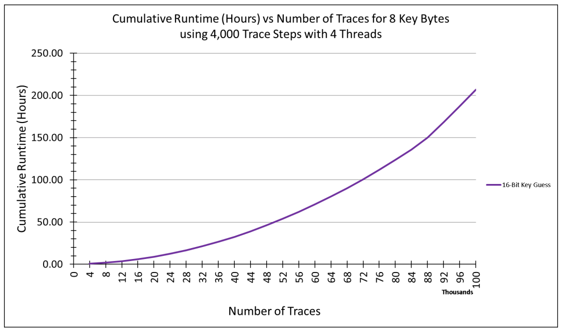


Figure 6.42: Cumulative runtime of 16-bit CPA selection function key guess size using 2 rounds of Keccak.

Key2 - Attacked 5 Key Lanes (40 Bytes) Using 2 Rounds of Keccak

After the successful attack of the first key lane of Key2, the next step is to attack all five key lanes which are on the bottom plane of the Keccak state. The CPA attack against these lanes for the 40 key bytes utilizes only 2 rounds of Keccak and the 8-bit selection function key guess size against the target operation. The success rate for recovering these 40 key bytes is expressed in Figure 6.43. The data results for the runtime and cumulative runtime of the 8-bit CPA selection function key guess size are expressed in Figure 6.44 and Figure 6.45, respectively. The data of these curves show that attacking 40 key bytes with the 8-bit CPA selection function key guess size is feasible, especially if one is concerned with the runtime, then it takes less than 40 minutes to complete an attack for 40 key bytes.

The summary of the number of successfully recovered key bytes can be seen in Figure 6.46 where the correct recovered key bytes are shaded in green and incorrect recovered key bytes are shaded in red. Compared to the attack of 40 key bytes of Key1 where the CPA algorithm is able to recover only 35 of 40 key bytes correctly, which is a success rate of 87.5%, the CPA is able to recover 38 of 40 key bytes of Key2 correctly. That is a success rate of 95% that can be confirmed in the Key2 success rate plot in Figure 6.43. Similar to the attack result with Key1, the incorrect recovered key bytes happen on the last key lane. Looking at the ranked top-8 correlation values of that lane as is illustrated in Figure 6.47, the correlation of the correct key guess for the targeted key byte is not that far off such that it is, at most, 1 below the top correlation value. By appyling more traces, eventually the correct key guess will bubble to the top as the one with the highest correlation value, which means more intensive correlation computation between the power model and the power traces leading to longer runtime.

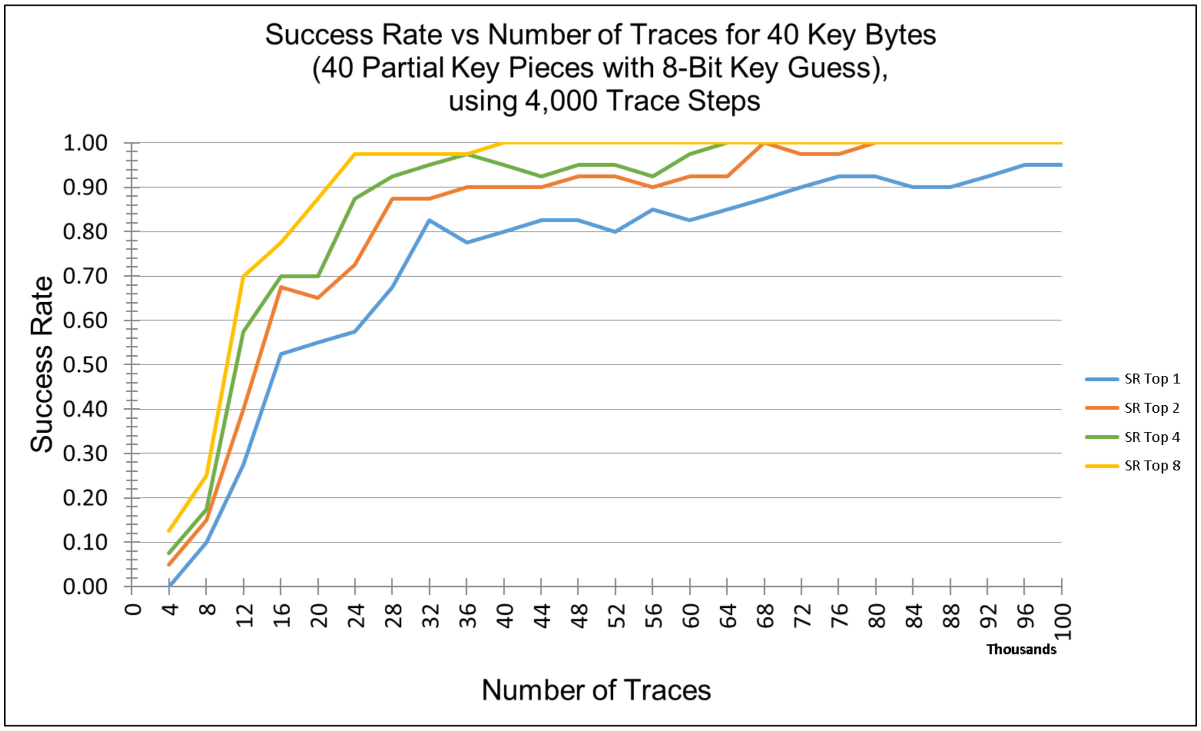


Figure 6.43: Success rates of 8-bit CPA selection function key guess size attacking 40 key bytes using 2 rounds of Keccak.

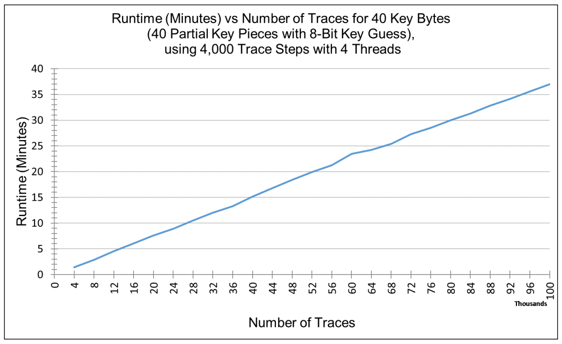


Figure 6.44: Runtime of 8-bit CPA selection function key guess size using 2 rounds of Keccak.

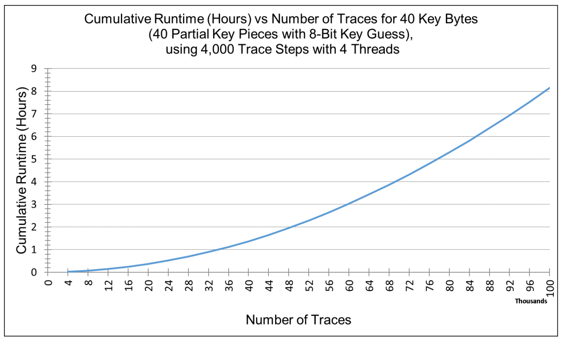
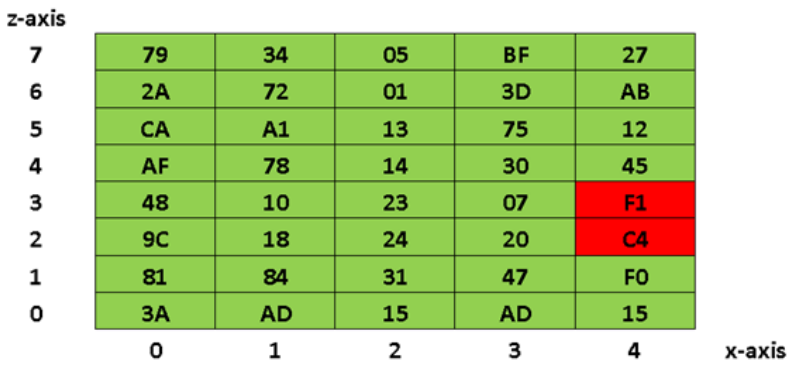
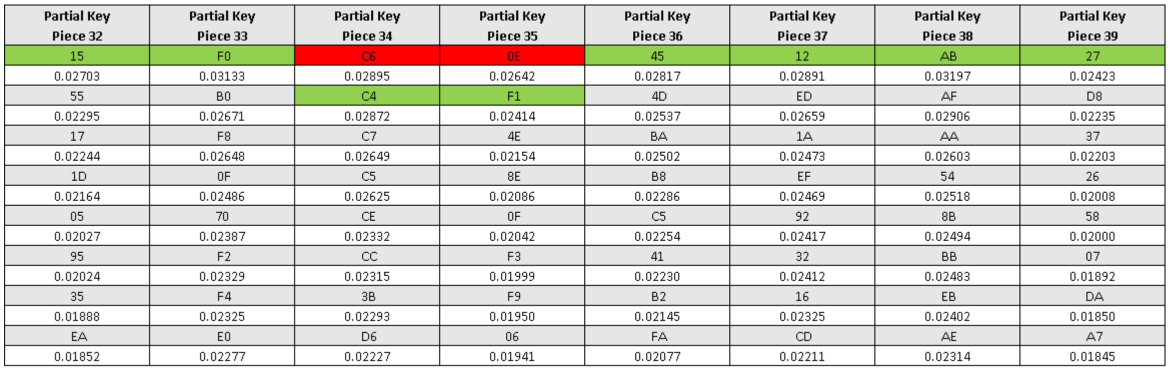


Figure 6.45: Cumulative runtime of 8-bit CPA selection function key guess size using 2 rounds of Keccak.



z-axis					
7	79	34	05	BF	27
6	2A	72	01	3D	AB
5	CA	A1	13	75	12
4	AF	78	14	30	45
3	48	10	23	07	F1
2	9C	18	24	20	C4
1	81	84	31	47	F0
0	3A	AD	15	AD	15
	0	1	2	3	4	x-axis

Figure 6.46: Recovered 38 of 40 key bytes of Key2 using 8-bit CPA selection function key guess size.



Partial Key Piece 32	Partial Key Piece 33	Partial Key Piece 34	Partial Key Piece 35	Partial Key Piece 36	Partial Key Piece 37	Partial Key Piece 38	Partial Key Piece 39
15	F0	C6	0E	45	12	AB	27
0.02703	0.03133	0.02895	0.02642	0.02817	0.02891	0.03197	0.02423
55	B0	C4	F1	4D	ED	AF	D8
0.02295	0.02671	0.02872	0.02414	0.02537	0.02659	0.02906	0.02235
17	F8	C7	4E	BA	1A	AA	37
0.02244	0.02648	0.02649	0.02154	0.02502	0.02473	0.02603	0.02203
1D	0F	C5	8E	B8	EF	54	26
0.02164	0.02486	0.02625	0.02086	0.02286	0.02469	0.02518	0.02008
05	70	CE	0F	C5	92	8B	58
0.02027	0.02387	0.02332	0.02042	0.02254	0.02417	0.02494	0.02000
95	F2	CC	F3	41	32	BB	07
0.02024	0.02329	0.02315	0.01999	0.02230	0.02412	0.02483	0.01892
35	F4	3B	F9	B2	16	EB	DA
0.01888	0.02325	0.02293	0.01950	0.02145	0.02325	0.02402	0.01850
EA	E0	D6	06	FA	CD	AE	A7
0.01852	0.02277	0.02227	0.01941	0.02077	0.02211	0.02314	0.01845

Figure 6.47: Top-8 correlation values of partial key pieces of Key2 in lane 4 of the bottom plane. Partial key pieces are in bytes and represented as hex values.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This work took an in-depth look at the application of the power analysis attacks that use the n-bit CPA selection function key guess technique to extract the key from the MAC-Keccak using simulated power traces that no other research institutions have done before. The research leveraged the simulated power trace methodology from [9] in order to successfully develop, synthesize and simulate Keccak high-speed core hardware implementation from the SHA-3 competition with the Synopsys 130-nm CMOS standard cell library to generate power waveforms for power analysis attacks.

The work provided a robust and re-usable framework for the study and development of the power analysis attacks. The re-usable framework was designed and written in a modular fashion to be flexible to attack both simulated and physical power traces of AES, MAC-Keccak and future crypto systems. The robust design of the framework allows it to be easily adaptable for future research.

From the generation and application of the simulated power traces, this research successfully attacked unprotected MAC-Keccak with keys $<= 40$ bytes and extracted the keys by using 1-bit, 2-bit, 4-bit, 8-bit and 16-bit CPA selection function key guess size attacks from the exploitation and targeting of the Keccak first round θ function that absorbed the "key||message". As the results of the n-bit CPA selection function key guess size attacks, it was observed the larger the selection function key guess size used, the better the signal-to-noise-ratio (SNR), therefore requiring fewer numbers of traces needed to retrieve the key but suffering from higher computation time. With 16-bit CPA selection function key guess size attack using 2 rounds of Keccak, 50K traces and 7 hours of computation were required for 100% key extraction of a single key lane of 8 bytes. Compared to larger selection function key guess size, smaller selection function key guess size has lower SNR that requires higher number of applied traces for successful key extraction and utilizes less computational time due to smaller number of guesses to iterate through. With 4-bit CPA selection function key guess size attack using 2 rounds of Keccak, it requires 200K traces and 7 minutes of computation for 100% key extraction of a single key lane. Of all the n-bit CPA selection function key guess sizes used, 8-bit CPA selection function key guess size is the best candidate in terms of computation time and the number of required traces for mounting successful attacks. With 8-bit CPA selection function key guess size attack using 2 rounds of Keccak, 80K traces and 7 minutes of computation were required for 100% single key lane extraction.

Finally, the research also explored and analyzed the attempted method of n-bit x n-bit cascade θ_{plane} for attacking the first and second planes of the 3D Keccak state where the key expanded beyond 40 bytes. The n-bit x n-bit cascade θ_{plane} attack method leveraged and expanded from the successful attack where the key is only on the bottom plane. The usage of power model of the first round θ function for n-bit x n-bit cascade θ_{plane} attack is proven not to be sufficient for attacking due to simple XOR operations and low correlation values. Therefore, MAC-Keccak utilizing keys greater than 40 bytes such that the key bytes beyond the first plane is safe from first round θ function power model CPA attacks.

7.2 Future Work

There are several avenues to extend this thesis work. Since this research focused specifically on attacking hardware implementation using simulated power consumption waveforms to extract the key from the MAC-Keccak and showed that a 1st plane key is prone to CPA attacks, future work may use the same attack methodology and focus on physical power extraction and analysis. One could implement the same Keccak high-speed core architecture as used in this thesis on the FPGA development board and perform the first round θ power model CPA attacks. This way it will be interesting to compare and analyze the attack framework computation performance between simulated and physical power trace CPA attacks. Due to the fact physical traces are susceptible to the device and environment noise and measurement error, the physical attack may require a significantly larger number of power traces for successful attacks, resulting in a more computationally expensive attack. Future work could leverage the physical FPGA framework from Kevin Meritt in [12] that was used for attacking physical AES traces and modify it for physical attacks of MAC-Keccak by replacing the AES crypto core for Keccak high-speed core, and modifying PC GUI interactions with the FPGA board for sending down messages and collecting power traces. Once the physical traces are generated, one could use the attack framework in this thesis with the current MAC-Keccak predefined power model or user's defined power model with high confidence that the framework will work with the physical traces since the framework has been tested with the AES physical power traces from Meritt in [12] successfully.

Another idea for future projects to expand from this thesis is to implement countermeasures such as blinding/masking to deter first round θ power model CPA attacks. One could implement the three-share masking technique proposed by Keccak's developers for the Keccak high-speed core in [16] for simulated or physical power trace generation. It would be interesting to see if the proposed masking technique could withstand CPA attacks with ideal simulated traces that do not exhibit noise and measurement error such as from physical devices. Other parameters can be to see if the three-share masking can deter the attacks by looking at the number of applied power traces and also the n-bit CPA selection function key guess sizes.

The work in this thesis applies the side channel vulnerability assessment of the Keccak high-speed core hardware implementation using gate-level circuit-simulated power consumption waveforms from Synopsys EDA tools with a 130-nm CMOS standard cell library. The success of the CPA attack depends on how accurate the power model, i.e. using Hamming weight versus Hamming distance, is for the target operation to correlate with the power traces. As mentioned earlier the total power consumption of the CMOS circuit is the sum of the static power and dynamic power. Potential future work is to investigate the effect of using smaller technology sizes such as 45nm, 16nm or 7nm CMOS standard cell library and see how that will impact the CPA attack. As the technology gets smaller in size, the static power is exponentially increasing whereas the dynamic power is linearly decreasing. It will be interesting to see if the dynamic power for these smaller size technologies is overshadowed by the constant larger static power, that is using and realizing them in existing power leakage models like Hamming weight or Hamming distance in CPA attacks to be in any way effective as with larger technology sizes.

Finally, another possibility for future work is to investigate other potential targets of the Keccak operations for better power correlations. This thesis only focused on the θ_{plane} calculations of the first round θ operation that performed column XOR operations. Since the θ operation is a linear function, it does not produce high correlation values such that this thesis is only able to extract the key from MAC-Keccak on the first plane, i.e. < 40 bytes. From past experience with attacking the non-linear operation such as the S-Box of AES that gave the strongest correlation values, one would want to target the non-linear operation of Keccak. In this case, future work should look at the first round χ non-linear operation for high correlation values. Besides targeting the first round χ non-linear operation, another target is to examine the output of the first round as a result of the five Keccak operations. This way the output of the first round is known to have gone through the non-linear operation and, hopefully, will provide better correlation values to attack key bytes beyond the first plane.

Bibliography

[1] A. Kaminsky, "CSCI 462 - Introduction to Cryptography, Chapter 11. Hash Functions, Lecture Notes." [Online]. Available: http://www.cs.rit.edu/ ark/spring2015/462/module11/notes.shtml, March 2015.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "The Keccak reference, Version 3.0." [Online]. Available: http://keccak.noekeon.org/Keccak-reference-3.0.pdf, January 2011.

[3] NIST, "DRAFT FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," *Federal Information Processing Standards Publication*, 2014.

[4] M. Taha and P. Schaumont, "Side-Channel Analysis of MAC-Keccak," in *Signal Processing Workshop on Higher-Order Statistics (SPWHOS)*, pp. 125–130, 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 2013.

[5] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Cryptographic sponge functions." [Online]. Available: http://sponge.noekeon.org/CSF-0.1.pdf, 2011.

[6] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO'99, pp. 388–397, London, UK: Springer-Verlag, 1999.

[7] G. Smith, "Power Analysis Attacks on the SHA-3 Candidate Grøstl," Master's thesis, Rochester Institute of Technology, 2012.

[8] J. P. Uyemura, *Introduction to VLSI Circuits and Systems*. John Wiley & Sons, second ed., 2002.

[9] K. J. Smith, "Methodologies for Power Analysis Attacks on Hardware Implementations of AES," Master's thesis, Rochester Institute of Technology, 2009.

[10] CRI, "DPA Workstation Training: Differential Power Analysis," Cryptographic Research Inc., January 2012.

[11] N. Benhadjyoussef, H. Mestiri, M. Machhout, and R. Tourki, “Implementation of CPA analysis against AES design on FPGA,” in *Communications and Information Technology (ICCIT), 2012 International Conference on*, pp. 124–128, June 2012.

[12] K. Meritt, “Differential Power Analysis Study and Experimental Results,” Master’s thesis, Rochester Institute of Technology, 2012.

[13] F. Zhang and Z. J. Shi, “Differential and Correlation Power Analysis Attacks on HMAC-Whirlpool,” in *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pp. 359–365, April 2011.

[14] M. Taha and P. Schaumont, “Differential Power Analysis of MAC-Keccak at Any Key-Length,” in *8th International Workshop on Security (IWSEC2013)*, pp. 68–82, 2013.

[15] P. Luo, Y. Fei, X. Fang, A. A. Ding, M. Leeser, and D. R. Kaeli, “Power Analysis Attack on Hardware Implementation of MAC-Keccak on FPGAs,” in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–7, 2014.

[16] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, “The Keccak implementation overview, Version 3.2.” [Online]. Available: http://keccak.noekeon.org/Keccak-implementation-3.2.pdf, May 2012.

[17] NIST, “Cryptographic Hash and SHA-3 Standard Development.” [Online]. Available: http://csrc.nist.gov/groups/ST/hash/index.html, 2013.

[18] M. Bellare, R. Canetti, and H. Krawczyk, “Keying Hash Functions for Message Authentication,” in *Advances in Cryptology CRYPTO 96*, vol. 1109 of *Lecture Notes in Computer Science*, pp. 1–15, Springer Berlin, Heidelberg, 1996.

[19] O. Benoit and T. Peyrin, “Side-channel Analysis of Six SHA-3 Candidates,” in *Cryptographic Hardware and Embedded Systems, CHES 2010*, vol. 6225 of *Lecture Notes in Computer Science*, pp. 140–157, Springer Berlin, Heidelberg, 2010.

[20] J.-J. Quisquater, “Side channel attacks: State-of-the-art.” [Online]. Available: http://www.ipa.go.jp/security/enc/CRYPTREC/fy15/doc/1047_Side_Channel_report.pdf, 2002.

[21] M. Aigner and E. Oswald, “Power Analysis Tutorial,” Institute for Applied Information Processing and Communication, University of Technology Graz - Seminar, Tech. Rep., 2000.

[22] F.-X. Standaert, “Introduction to Side-Channel Attacks,” in *Secure Integrated Circuits and Systems*, pp. 27–44, Springer, 2009.

[23] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, “Investigations of Power Analysis Attacks on Smartcards,” in *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, pp. 17–27, Berkeley, CA, USA: USENIX Association, 1999.

[24] E. Peeters, F.-X. Standaert, and J.-J. Quisquater, “Power and Electromagnetic Analysis: Improved Model, Consequences and Comparisons,” in *Integration, the VLSI Journal*, vol. 40, pp. 52–60, Spring 2007.

[25] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, “Examining Smart-Card Security under the Threat of Power Analysis Attacks,” in *IEEE Transactions on Computer*, vol. 51, pp. 541–552, 2002.

[26] E. Brier, C. Clavier, and F. Olivier, “Correlation Power Analysis with a Leakage Model,” in *CHES 2004*, vol. 3156 of *Lecture Notes in Computer Science*, pp. 135–152, 2004.

[27] A. Papoulis and S. U. Pillai, *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, fourth ed., 2002.

[28] E. Brier, C. Clavier, and F. Olivier, “Optimal statistical power analysis,” Cryptology ePrint Archive, Report 2003/152, 2003.

[29] R. McEvoy, M. Tunstall, C. C. Murphy, and W. P. Marnane, “Differential Power Analysis of HMAC based on SHA-2, and Countermeasures,” in *Proceedings of the 8th International Conference on Information Security Applications*, WISA’07, pp. 317–332, Berlin, Heidelberg: Springer-Verlag, 2007.

[30] M. Zohner, M. Kasper, M. Stottinger, and S. A. Huss, “Side Channel Analysis of the SHA-3 Finalists,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1012–1017, 2012.

[31] B. Jungk and J. Apfelbeck, “Area-efficient FPGA Implementations of the SHA-3 Finalists,” in *2011 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 235–241, 2011.

[32] D. R. Stinson, *Cryptography: Theory and Practice*. Chapman and Hall/CRC, third ed., 2006.