Quantifying Timing-Based Information Flow in Cryptographic Hardware

Baolei Mao*, Wei Hu†, Alric Althoff‡, Janarbek Matai‡, Jason Oberg†, Dejun Mu*, Timothy Sherwood† and Ryan Kastner†

*Northwestern Polytechnical University, PR China
Email:{maobaolei, mudejun}@nwpu.edu.cn
†University of California, San Diego
Email: {weh040, aalthoff, jmatai, jkoberg, kastner}@ucsd.edu
‡University of California, Santa Barbara
Email: sherwood@cs.ucsb.edu

Abstract—Cryptographic function implementations are known to leak information about private keys through timing information. By using statistical analysis of the variations in runtime required to encrypt different messages, an attacker can relatively easily determine the key with high probability. There are many mitigation techniques to combat these side channels; however, there are limited metrics available to quantify the effectiveness of these mitigation attacks. In this work, we employ information theoretic ideas to quantify the amount of leakage that can be extracted from runtime measurements and reveal the influence of individual key bits on the timing observations across a variety of hardware implementations. By studying different RSA hardware architectures (each with different performance optimizations and mitigation techniques), we determine the effectiveness of these information theoretic techniques against the success of attacks. Our experimental results show that mutual information is a promising metric to quantify timing-based information leakage and it also correlates to the attack-ability of a cryptographic implementation.

I. INTRODUCTION

Kocher’s seminal work on side channels demonstrates that the time to compute a cryptographic function leaks a significant amount of information about the secret key [1]. The attack is simple and elegant. The attacker provides different plain text messages to the cryptographic function and measures the amount of time that it takes to perform the encryption. Using knowledge of the cryptographic algorithm and statistical techniques, the attacker can guess with high certainty the individual bits of the secret key, which renders functionally strong cryptographic algorithms useless.

The initial timing side channel attacks target timing variations due to properties of the software cryptographic implementation [1], [2], e.g., variable computational delay between conditional branches that depend upon the value of the key bits. A common countermeasure focuses on “fuzzing” the timing signal by performing input blinding or forcing the implementation to run in a random or constant time [1], [3]. This decouples the relation between the key bits and runtime observations, which would otherwise be easily captured by statistical analysis tools.

While there are numerous attacks and mitigation techniques focusing on software timing channels, hardware timing channels are less studied. Ciet et al [4] used a parallel implementation of RSA with Residue Number Systems to resist side channel attacks. Oberg et al [5] provide a framework for detecting timing channels, and verifying timing channels in designed caches, encryption cores, and interconnect networks are eliminated. Regardless, timing attacks remain robust even when applying mitigation techniques. Unless you are willing to go to great lengths to eliminate the side channel, there will be leakage. And given enough information, secret information can be extracted. Therefore, a method to quantify the amount of leakage is important to understand the risks inherent in the hardware implementation.

Claude Shannon pioneered the notion of information theory during World War II in order to measure channel capacity of a transmitting medium. He developed the mathematical foundation to answer the question: “How much data could we maximally expect to transmit across the channel?” The underlying metrics that he devised including entropy and mutual information; they play a fundamental role on deriving these theoretical limits. In this work, we investigate how these same information theoretic metrics can be used on timing side channel between the encryption key and the time to compute the corresponding cryptographic function.

Mutual information measures the dependence between two random variables. In the context of a timing channel, the mutual information between the key and the time to compute the cryptographic function bounds the number of key bits that is leaked through runtime observations. In theory, a high mutual information between these two variables indicates the strength of the timing side channel. Indeed, Köpf studied the amount of information leakage versus performance and provided bounds on the information flow [6], [7]. However, to the best of our knowledge there is few work that determines whether mutual information is a good metric for quantifying the amount of leakage through the timing channel for given hardware architectures. Nor is there work that shows any relation between the amount of information leakage and the attack-ability. This work provides a study on both of these ideas.

In this work, we first show how secret key information of a RSA core can be efficiently retrieved from runtime measurements. We provide a methodology for quantifying the information leakage of cryptographic hardware architectures
designed using different optimization techniques and timing leakage mitigation strategies. We show that mutual information provides key insights into the amount of information that a particular hardware architecture provides about the secret key. Using RSA as an exemplar application, we show how mutual information can be used to quantify the amount of information leaked through the timing side channel. This is done by analyzing five different hardware implementations of RSA cores with performance optimization and another five with different mitigation techniques.

Our methods provide a better understanding of hardware timing information flows, and they reveal the effects of architectural optimizations and mitigation techniques on timing channels. Specifically, this paper makes the following contributions:

- Proposing a metric that enables designers to reason about the security of their hardware design with respect to timing side channels;
- Demonstrating how information theoretic methods such as entropy and mutual information quantify timing leakage in different hardware architectures of RSA;
- Presenting experimental results that reveal the relationship between different timing channel mitigation techniques and the reduction in leakage using information theoretic measures.

The reminder of this paper is organized as follows. Section II describes the threat model. In section III, we cover the basics of the RSA algorithm, the core ideas behind the statistical and information theoretical measures that we use as security metrics; we describe Kocher’s method to attack a hardware implementation of the basic RSA algorithm. Section IV discusses the effects of logic optimization and mitigation techniques. Section V gives experimental results. We briefly review the related work in Section VI. We conclude in Section VII.

II. Threat Model

We consider hardware components where the attacker aims to gather critical information through a timing channel. We use the RSA cryptographic function throughout the paper, but the idea can extend to other components where the pertinent information is a function of the computation time.

We assume that the attacker can determine the amount of cycles for each execution of the hardware component. That is, the attacker can determine the cycle in which the execution begins and ends. In general, timing side channel attacks are relatively robust to measurement noise, i.e., it is not necessary to know the exact number of cycles. However, we do not add any sort of noise into the timing measurements in our experiments.

Finally, we have control over some of the inputs to the function. For example, in the cryptographic function example, we can change the plaintext, and determine the number of cycles that the function requires to compute the ciphertext. Note that it is common for cores to have input and output ports that denote when the function is completed, and when the component is capable of taking new data. These are used for synchronization between components. Our experiments use these ports to determine the number of cycles. Furthermore, we assume that we can distinguish between the time of different executions, e.g., in cases where the hardware performs multiple encryptions in a parallel or pipelined manner.

Our experiments are performed on an FPGA. We create different cryptographic function implementations, and add control logic around these cores that allows us to provide it with different input data, and determine the amount of cycles required to compute the function for each different set of inputs. The threat model extends beyond this experimental setup. For example, it is also relevant to systems that provide debug interfaces that give timing information on the different subcomponents.

III. Preliminaries

This section starts by showing the existence of a timing channel in the basic RSA cipher. Then we cover some statistical analysis tools for understanding timing information flows and retrieving secret information from runtime measurements. Finally, we will briefly review Kocher’s timing attack method on RSA implementation [1].

A. Basic RSA Algorithm

We focus our analysis on the RSA algorithm since it is known to have a timing side channel, and it has a number of attacks and mitigation techniques. Thus, it allows us to analyze the effectiveness of the information theoretic techniques as a metric for timing leakage.

RSA is a public key cipher that maintains a key pair for encryption and decryption. Given a public key $e$, secret key $d$, modular $n$, plain text $m$, the cipher text $c$ is encrypted and decrypted as follows:

$$c = m^e \mod n$$

$$m = c^d \mod n$$

Equation (1) shows modular exponentiation, which is the basic operation of RSA. Taking the decryption process as an example, Algorithms 1 and 2 illustrates how modular exponentiation is calculated through repeated square-and-multiply from right-to-left and left-to-right respectively.

### Algorithm 1 Modular exponentiation $c^d \mod n$ calculated using square-and-multiply (right-to-left)

1: $m[0] := 1$
2: $s[0] := c$
3: for $k := 0$ to $w − 1$ do
4:    if $d[k] == 1$ then
5:        $m[k + 1] := m[k] \cdot s[k] \mod n$
6:    else
7:        $m[k + 1] := m[k]$
8:    end if
9: $s[k + 1] := s[k] \mod n$
10: end for
11: Return $m[w]$
The above algorithms perform modular multiplication when the key bit under consideration is one (Lines 4-5 in Algorithm 1 and Lines 3-4 in Algorithm 2). By comparison, only a simple assignment is needed when the current key bit is zero (Lines 6-7 in Algorithm 1 and Lines 5-6 in Algorithm 2). Such runtime difference creates a timing channel that an attacker can use to ascertain information about the key. We will show how secret key can be recovered through simple yet effective statistical analysis in Section III-C.

In our successive discussions, we assume that the attacker knows the implementation details of the RSA algorithm, can specify messages and keys as algorithm input and has the ability to measure the total runtime for processing each message under a given key. We focus on 32-bit implementations of RSA for the ease of result interpretation though our method applies to RSA cores of arbitrary key length.

B. Definitions

Now we provide a quick overview of the statistical and information theoretic techniques that we use in the remainder of the paper. These are far from a complete description, and we encourage an interested reader to consult other sources for more information [8].

Variance is a measure of the relative distances of a set of numbers from their mean. It is a parameter that describes the probability distribution of the set of numbers. A small variance indicates that the data samples tend to distribute close to the mean, while a larger variance indicates that the data points spread out from the mean.

Given a set of observed samples \(x_1, x_2, \ldots, x_n\) of a random variable \(X\). The variance of the data set is:

\[
\text{var}(X) = \sum_{i=1}^{n} (x_i - \bar{X})^2
\]

where \(\bar{X}\) is the expected value of \(x_1, x_2, \ldots, x_n\). Given two independent random variables \(X\) and \(Y\), we have

\[
\text{var}(X \pm Y) = \text{var}(X) + \text{var}(Y)
\]

Entropy measures the uncertainty of a random variable. Using \(p(x)\) to denote the probability density function (pdf) of random variable \(X\), the Shannon Entropy is defined as:

\[
H(X) = -\sum_{x \in X} p(x) \log p(x)
\]

Given two random variables \(X\) and \(Y\), let \(p(X, Y)\) denote the pdf of their joint distribution. The Joint Entropy of \(X\) and \(Y\) is defined by (5).

\[
H(X, Y) = -\sum_{x \in X, y \in Y} p(x, y) \log p(x, y)
\]

Mutual Information quantifies the reduced uncertainty of random variable \(X\) when another random variable \(Y\) is known. It is a measurement of how much information variable \(Y\) contains about variable \(X\). The mutual information between \(X\) and \(Y\) is defined as:

\[
I(X; Y) = H(X) + H(Y) - H(X, Y), \quad x \in X, y \in Y
\]

where \(H(X|Y)\) is the Conditional Entropy of \(X\) given \(Y\).

With an understanding of the statistical and information theoretic metrics, now we describe Kocher’s timing attack method based on variance analysis.

C. Kocher’s Timing Attack

Kocher was the first to provide a comprehensive theoretical analysis on timing attack using variance analysis [1].

The attack is based on the assumption that the runtimes for processing different key bits are independent, i.e., given a number of messages, the runtime observations for different key bits compose independent random variables. Let \(T\) denote the vector that contains the total runtime observations for processing \(N\) messages and \(t_i (i = 0, 1, \ldots, w - 1)\) denote the vector that contains the runtime for the \(i\)-th key bit for processing \(N\) messages. Using (3), we have

\[
\text{var}(T) = \sum_{i=0}^{w-1} \text{var}(t_i)
\]

The attack makes guesses of the current key bit (assuming it could be either zero or one) and obtains the runtime vectors \(t_i^0\) and \(t_i^1\) through observation. For a correct guess \(t_i^c\) \((c \in \{0, 1\})\), \(\text{var}(T - t_i^c)\) will decrease \(\text{var}(T)\) by \(\text{var}(t_i)\); for the wrong guess, \(t_i^{1-c}\) will be independent from the correct runtime observations, and \(\text{var}(T - t_i^{1-c})\) should theoretically increase \(\text{var}(T)\) by \(\text{var}(t_i^{1-c})\) according to Equation (3).

In a real attack, the runtime observation vectors are not perfectly independent from each other due to the limited number of samples. However, a correct guess still tends to decrease the variance by a larger amount than a wrong guess. In the following section, we will perform attack on the basic RSA and show the intuition behind Kocher’s attack method.

We implement a 32-bit instance of RSA (Algorithm 1) on an FPGA and use variance analysis to perform timing attack. In our attack, only 4000 messages are tested, which is a considerably small portion (4000/2^{32} = 0.0001%) of the state space. Table I shows the remaining variances and the decrease in variance after each guess step. The bold italic numbers correspond to wrong key bit guesses.

As seen in Table I, the initial variance of the total runtime is 793.6. After the first key bit guess, the variance decreases by 32.3 to 761.3. After an additional guess, the variance further decreases to 759.7; the decrease in variance is only 1.6. For the
TABLE I. VARIANCE ANALYSIS USING KOCHER’S TIMING ATTACK METHOD.

<table>
<thead>
<tr>
<th></th>
<th>793.6</th>
<th>761.3</th>
<th>759.7</th>
<th>725.2</th>
<th>685.1</th>
<th>649.2</th>
<th>646.0</th>
<th>600.0</th>
<th>597.7</th>
<th>567.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Δ Var</td>
<td>32.3</td>
<td>1.6</td>
<td>34.5</td>
<td>0.0</td>
<td>40.2</td>
<td>35.9</td>
<td>31.1</td>
<td>45.4</td>
<td>2.9</td>
<td>30.3</td>
</tr>
</tbody>
</table>

Bits 10 - 20

| Variance | 535.0 | 495.0 | 495.0 | 451.1 | 413.3 | 365.8 | 368.5 | 367.2 | 322.7 | 291.4 |
| Δ Var   | 32.4  | 40.0  | 0.0   | 44.0  | 37.7  | 0.0   | 47.5  | 0.0   | 4.7   | 38.5  |
| Inst & Bits 0 - 9 |       |       |       |       |       |       |       |       |       |       |

Bits 21 - 31

| Variance | 257.1 | 224.0 | 225.7 | 189.9 | 154.8 | 117.2 | 86.49 | 83.60 | 52.44 | 52.44 |
| Δ Var   | 34.3  | 33.1  | 0.4   | 33.7  | 35.1  | 37.6  | 30.7  | 2.9   | 31.2  | 0.0   |

2 pipeline unroll

From Table I, the runtime difference between different algorithmic branches in the basic RSA makes it vulnerable to simple yet effective timing attack. Among the 32 key bits, 26 are successfully recovered after testing only 4000 messages. In the following section, we show how different hardware RSA architectures are vulnerable/resistant to timing attacks. Specifically, we generate a number of RSA architectures for timing attack through logic optimization in Section IV-A and build several mitigation techniques into hardware RSA implementations in Section IV-B.

IV. RSA ARCHITECTURES

A. RSA Performance Optimizations

Hardware optimizations have a significant impact on timing. For example, architectures exploiting significant parallelism or pipelining require a smaller number of clock cycles than those that work in a sequential manner. We are interested in understanding if and how these different optimizations affect the amount of information leaked through their runtime.

High-level synthesis (HLS) is a design methodology that can be used to generate different hardware implementations from high-level language algorithm specifications. This allows quick profile of various architectures of the same algorithm. Optimizations are typically specified using pragmas that tell the HLS tool how to optimize particular regions of the code, e.g., pipeline, and unroll. In this work, we use Xilinx Vivado HLS tool to generate five unique 32-bit RSA architectures using different optimization strategies.

TABLE II. FIVE UNIQUE RSA ARCHITECTURES GENERATED USING DIFFERENT HLS OPTIMIZATIONS. ’-’ INDICATES NO OPTIMIZATION FOR THAT LOOP.

<table>
<thead>
<tr>
<th>Designs</th>
<th>L1</th>
<th>L2</th>
<th>Avg. Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>-</td>
<td>-</td>
<td>3023</td>
</tr>
<tr>
<td>Pipeline</td>
<td>-</td>
<td>pipeline</td>
<td>1646</td>
</tr>
<tr>
<td>Unroll</td>
<td>-</td>
<td>unroll</td>
<td>2699</td>
</tr>
<tr>
<td>Pipeline&amp;unroll_1</td>
<td>-</td>
<td>pipeline unroll</td>
<td>665</td>
</tr>
<tr>
<td>Pipeline&amp;unroll_2</td>
<td>-</td>
<td>pipeline unroll</td>
<td>1475</td>
</tr>
</tbody>
</table>

Design Original does not have any optimization; it is largely sequential. The second design pipelines the modular multiply (L2). Design Unroll fully unrolls the modular multiply loop. Design Pipe&unroll_1 partially unrolls and pipelines the modular multiply loop. Design Pipe&unroll_2 unrolls the modular multiply loop and pipelines the modular exponentiation loop.

B. RSA Timing Mitigation Techniques

As we will show in the experimental results section, different performance optimizations will change the amount of time to execute the function, which may make the resulting design easier or harder to attack. However, it may not completely eliminate the timing channel.

Mitigation techniques make changes in the algorithm itself in order to reduce timing-based leakage. These mitigation techniques typically fall to two categories, either making runtime measurements constant or random [3], [9]. The simplest way to hide timing variations from observations is to make the total runtime for encrypting all messages constant [3]. This
completely eliminates the timing side channel. However, it comes at a high performance penalty since all executions will have the worst case execution time. A more intelligent alternative is to quantize all RSA computations. Encryption times are bounded to multiples of some predefined time quantum [10]. This method can help reduce computation cost but cannot fully eliminate timing leakage. Other possible defenses attempt to decouple runtime measurements from plain texts. These include performing dummy modular multiplication even when the key bit is zero [3], moving the modular square into the conditional statements [3], inserting additional reduction in the Montgomery algorithm even if unnecessary [9], and introducing some random number into RSA computation (i.e., RSA blinding) to make the runtime observation unrelated to the plain text [3].

The timing leakage is primarily caused by latency difference when the encryption takes different algorithmic flows (see Algorithms 1 and 2). Thus, minimizing the timing difference of the different conditional branches provides a way to mitigate the timing channel. Other approaches keep the latency difference there but focus on the attack method, e.g., making the total runtime independent of the message. In our analysis, we consider several RSA implementations with built-in mitigation techniques. These designs include constant runtime (Base design), Left-to-right multiply always, Power ladder and Montgomery multiplication. Figure 2 shows the algorithm details of these different architectures.

The **Base** design selects a maximum runtime that is safe to complete encryption for all messages and key pairs, inserts additional delays beyond the actual runtime needed for encrypting each message, and thus makes the runtime observations invariant to either the key or the plain text. The **Left-to-right multiply always** algorithm inserts a dummy multiply in the else statement of the conditional branch. This reduces the key dependent delay difference and helps mask the timing feature that causes key leakage. The **Power ladder** algorithm carefully re-designs the algorithmic flow. It moves the modular square operation into the conditional branch and always performs both modular multiply and square regardless of the current key bit. For these previous two architectures, the runtime of a modular multiplication operation is not constant; there is timing difference for different messages. The Montgomery multiplication algorithm uses a different modular multiplier (i.e., MontMult). The runtime of a modular multiplication operation using this new multiplier is totally determined by the modulus. Although there is still a timing difference caused by the conditional branch in the algorithm flow, it eliminates the timing difference resulting from different messages. Thus, the variance of the runtime is constantly zero for both key bit guesses, making it impossible to decide the correct key bit. Table III summarizes the different designs with mitigation techniques used in our analysis. In the next section, we will show how these different optimizations and mitigation techniques affect the amount of information leaked due to timing.

### Table III. RSA architectures with different mitigation techniques.

<table>
<thead>
<tr>
<th>Designs</th>
<th>Mitigation Technique</th>
<th>Avg. Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>Const. total runtime</td>
<td>4500</td>
</tr>
<tr>
<td>L-2-R always</td>
<td>Dummy modular multiply</td>
<td>1975</td>
</tr>
<tr>
<td>Power ladder</td>
<td>Re-design algorithm flow</td>
<td>2099</td>
</tr>
<tr>
<td>Montgomery</td>
<td>Const. time modular multiply</td>
<td>1768</td>
</tr>
</tbody>
</table>

V. EXPERIMENTAL RESULTS

A. Experimental Setup

We implement our timing attack framework using three modules: test vector generation, statistical analysis, and RSA timing. Figure 3 illustrates our experimental setup. In the test vector generation module, we use OpenSSL to generate RSA key-pairs (key and modulus) of specific length and produce random messages with Python’s pseudo-random number generator. In the statistical analysis module, we compute the variances to guess each key bit using Kocher’s timing attack method. We calculate the mutual information between the key bits and the total runtime to quantify information leakage. We implement the different RSA architectures on a Xilinx VC707 FPGA board.

Fig. 2. Algorithmic flows of RSA architectures with different mitigation techniques.

**Algorithmic Flows of RSA Architectures**

<table>
<thead>
<tr>
<th>Base Design</th>
<th>Left to Right Multiply Always</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Algorithmic Flows" /></td>
<td><img src="image" alt="Algorithmic Flows" /></td>
</tr>
</tbody>
</table>

Fig. 3. Framework of timing attack. We implement the RSA core on an FPGA, and use serial to transmit information to and from the FPGA.

B. Design Optimized for Performance

For the first set of experiments, we use five different 32-bit RSA cores optimized for performance (see Table II in Section IV-A). We profile runtime samples of 1000 different 32-bit key pairs for each of the five different architectures. In order to understand the relationship between hardware architecture and key information leakage, we perform an analysis based on Equation (6):

\[
I(k_i; T) = H(k_i) + H(T) - H(k_i, T)
\]

where \(k_i\) denotes the value of the \(i\)-th key bit and \(T\) is the observed encryption time for the entire key. This shows how much information leaks from the \(i\)-th key bit when encryption...
time is known. That is, how much does the total runtime depend on the \(i\)-th bit of the key. As we will later show, this mutual information \(I(k_i; T)\) provides a measure of the timing side channel.

Figure 4 shows the experimental results. We can see that Original architecture leaks about 0.6 bits of information on average while the Unroll architecture shows a little less key bit leakage than the Original design. The Pipeline design reduces information leakage to 0.4. Pipeline\&unroll_1 further reduces this leakage to around 0.2; this design results in constant time to perform the modular multiply though it still leaks information through control that is a function of the message. All these designs are implemented using one modular multiply function, so multiply and square operations execute sequentially. The pipeline and unroll directives in high level synthesis change the structure of inner-loop, which reduces key information leakage. The general trend is that the more parallelism, the lower the amount of leakage. Or conversely a sequential implementation leaks more information than a parallel one.

Design Pipeline\&unroll_2 leaks almost no key bit information except for the most significant bits. This design has a special architecture – the synthesis tool generates two modular multipliers, one for modular multiply and one for the modular square. These two modular multipliers run in parallel for each key bit iteration, and each iteration finishes in the same number of clock cycles. In addition, their modular multiply time and control logic time is constant, diminishing the effect from different messages. We can see that for most of the key bits, the leakage is very close to 0 but the curve increases dramatically at the end. That is due to the fact that architecture stops its execution after it reaches the most significant ‘1’ bit of the key. Thus, we can determine with great accuracy where this one ‘1’ bit resides due to the overall runtime of the algorithm. For example, the longest runtime will have a ‘1’ bit in the most significant bit. If that most significant bit is ‘0’, then it will have a shorter runtime since the algorithm will terminate sooner.

Another way of viewing this is using Equation (6) to derive the following:

\[
I(k_i; T) = H(T) - H(T|k_i)
\]

The entropy of the total runtime \(H(T)\) is constant; the decrease in conditional entropy \(H(T|k_i)\) contributes to the increase in \(I(k_i; T)\). While the decrease in the conditional entropy means that the uncertainty of total runtime \(T\) given \(k_i\) decreases. In other words, the high key bits have a more significant effect on \(I(k_i; T)\) by dominating the total runtime.

Note that there is no information leakage for the first two key bits in any of the architectures. This is due to the fact that these two bits are the same across all of the different keys due to requirements on how RSA keys are generated (i.e., they must be odd). Mutual information is symmetric, i.e., \(I(k_i; T) = I(T; k_i)\). Think about this in the opposite direction, i.e., how much can we learn about a constant bit from a runtime? – the answer is nothing. Therefore, \(I(k_i; T) = 0\).

We conduct a timing attack using Kocher’s method on each design. The attack results are shown in Figure 5. This shows the percentage of correct guesses using the attack for each key bit. Design Pipeline\&unroll_2 is difficult to attack; we are guessing around 50% correct, which is equivalent to randomly guessing the key bit. This should not be surprising given that this architecture has very little leakage as we discussed earlier. Correspondingly the mutual information in Figure 4 is mostly near 0. Note that the most significant bits have a slightly higher success rate in the attack, which corresponds to the strong up-tick in the mutual information at the most significant bits. Visually the remainder of the attacks follow a similar trend. We look more closely into the relationship between mutual information and the success of the attack in Section V-D.

C. Designs Using Mitigation Techniques

The second set of experiments focus on the architecture implemented with different mitigation techniques. These are originally discussed in Section IV-B and summarized in Table III and Figure 2. Assuming these mitigation techniques are effective, the mutual information for these should be lower than for those designed without mitigation, i.e., those designs described in the previous section. Then we profile runtime samples of the same 1000 different key pairs as Section V-B. And Figure 6 shows the mutual information results.

The Base design has a zero mutual information across all key bits. This is because the total runtime is constant. We cannot learn any information about the key upon observation of the runtime. The Left-to-right multiply always and Power ladder algorithms have mutual information of 0.2 bits and 0.3 bits, respectively. Montgomery multiplication reduces key information leakage to 0.1 bits. In this case the synthesis tool
generates one architecture with constant modular multiplication time and constant control logic time, removing the effect of different messages. But two modular multiply operations still run in a sequential way, so lower key bit leakage is not eliminated as is in design Pipeline\&unroll\_2.

Similar to the design Pipeline\&unroll\_2, several designs see an increase in mutual information for the higher key bits. The increase in mutual information $I(k_i;T)$ is caused by decrease in conditional entropy $H(T|k_i)$. This reveals that the higher key bits have a dominate effect on the total runtime. For designs that use either the left-to-right (Left-to-right multiply always and Power ladder designs) or right-to-left (Montgomery) algorithmic flow, the leftmost non-zero bit always determines the total number of algorithmic iterations and thus significantly influences the total runtime.

Then we conduct Kocher’s attack on these mitigation RSA designs, the results are shown in Figure 7. The designs with mitigation techniques are more difficult to attack than Left-to-right design. Power ladder design is easier to attack than Left-to-right multiply always design, but Montgomery and Base designs have success rate around 50\%, which is much less than Left-to-right multiply always and Power ladder designs.

When comparing these results with the mutual information results in Figure 6, we see two trends. First, a higher mutual information value indicates a higher likelihood of attack success. Second, there is an increase in the most significant bits in both mutual information results and attack success rate results for Power Ladder, Left-to-right multiply always and Montgomery designs. We will describe these relationship in next section using statistical analysis.

### D. Statistical Correlation

To better determine the connection between mutual information and leakage, we rely on Spearman’s $\rho$, a correlation measure between the sample means across all the bits. That is to say, $n^{-1} \sum_{i=0}^{31} I(k_i;T)$ and mean attack success rate (shown in Figure 5 and Figure 7). We take these means for each mutual information and for each success rate, yielding a pair for each design of the five mitigation and five performance optimized architectures. We justify using the arithmetic mean by testing the null hypothesis that the data are normally distributed using a Kolmogorov-Smirnov (KS) goodness-of-fit test after data standardization. While the mutual information is not strictly normally distributed we confirm that for bits 2–24 variation is well described by a Gaussian, though we do not exclude these data when computing correlation or significance. By failing to reject the null hypothesis of a KS test, we can safely assume that the arithmetic mean is a sensible measure of central tendency.

If for two length $n$ variables $X$ and $Y$, $x_i$ and $y_i$ are their values at position $i$, let $d_i = x_i - y_i$, $\rho$ is computed between two variables $X$ and $Y$ as:

$$\rho = 1 - 6 \sum_{i=1}^{n} \frac{d_i^2}{n(n^2 - 1)}$$

Spearman’s $\rho$ is nonparametric, which in this case means that the measure does not assume data comes from a particular distribution. This measure is well-suited to cases where the sample size is small. High correlation is achieved when one variable is a monotonic function of the other regardless of what this function may be. While significance of $\rho$ may be computed according to several distribution-based measures, we compute this value using an exact permutation test on account of the presence of a tie in the data between the means of the attack results. Briefly, a permutation test generates all non-redundant permutations of the variable with ties and reports as the $p$-value the exact probability that $\rho$ on data under permutation exceeds $\rho$ on the sample.

Using a one-tailed permutation test we see that $p = 0.018$. Therefore at the $\alpha = 0.05$ level we reject the null hypothesis that the mean mutual information and mean success rate are uncorrelated. This allows us to say that for a greater value of the mean mutual information we will see a greater mean success rate—indicating greater information leakage. In other words, both success rate and mutual information are able to describe timing leakage for cryptographic hardware architectures. Based on the timing channel leakage of these architectures, mutual information is positively correlated with the attack success rate.

### VI. RELATED WORK

There are numerous work that use information theoretic methods to ascertain the security of a system by analyzing the behavior of the software. Denning is amongst the first to relate security and information theory [11], using entropy to model relationships between statements in a program. McLean...
first describes the flow model security property [12]; it is later formalized quantitatively by Gray as an applied flow model, which relates noninterference to the maximum rate of flow between variables [13]. Clark et al. use different information theoretic measures to bound the information leaked from “while” programs [14]. Mica and Morgan use conditional entropy to calculate the channel capacity of a program [15]. McCamant and Ernst [16] present a technique to more precisely quantify how much information is revealed by the public output of C-like programs. Malacaria and Heusser [17] introduce quantitative information analysis for C code and show that the information leakage vulnerabilities in the Linux Kernel. Newsome et al. [18] use channel capacity as a quantitative measure of the influence of the inputs on the outputs of a program using x86 binaries. Information theory measures, e.g., the worst case mutual information [19] and min-entropy [20], are used at the system level to determine the difficulty of breaking into the system. None of these techniques deal with hardware designs as we describe in this work.

There are major efforts focus on using mutual information as a distinguisher function for differential power analysis. Gierlichs et al. [21] introduce this concept of mutual information analysis. They are inspired by Standaert et al. [22], who use mutual information to measure the amount of side-channel leakage for an implementation. Batina et al. [23] have a comprehensive study on mutual information analysis including the effects of estimating the probability distribution functions on the attacks. All of these use information theoretic metrics to attack the design, with a focus on power side channels. None of these works attempts to understand the effects of a particular hardware architecture or optimization on the side channel as we do in this work.

Perhaps the most similar work to ours is that done by Köpf et al. [6], [7]. They provide a bound on the information leakage through a timing channel based upon the number of observations. They use conditional entropy to derive that bound. This is similar in spirit to what we do in our work in that we are trying to derive a metric for security. However, we are looking at orthogonal variables – they look at the effect of the number of measurements on the leakage, while we are trying to understand how a design itself effects the leakage.

VII. CONCLUSION

In this paper, we study the potential for using the mutual information as a metric to quantify the amount of information a hardware architecture leaks through a timing channel. We design a number of different RSA hardware architectures that are optimized for performance and to mitigate the timing channels. We show that mutual information indicates lower leakage of data on those architectures using mitigation techniques. And we show that the mutual information and success of the attack is correlated, that is the higher the mutual information, the more likely that the attack will be successful. Our work shows that mutual information is a promising metric to quantify the information leakage through timing side channels.

REFERENCES


