COS 516 Final Project Report
Security Verification against Hardware Timing Side-Channel

Yung-Chin Chen, Yu-Wei Fan

December 12, 2024

1 Introduction

Hardware security has been of much interest in research since the two well-known attacks Meltdown [1] and Spectre [2] revealed serious security issues in modern processors in 2018. Unlike software security attacks such as SQL injection or cryptographic attacks that exploit the security vulnerabilities at the software level, Spectre and Meltdown instead utilize the fact that the underlying micro-architecture implementation of the processor is insecure. Conventionally, from the software security perspective, it is assumed that the underlying hardware is secure. Therefore, the system should be free from security attacks as long as the program is secure. However, Spectre and Meltdown exploit the micro-architectural performance feature: speculative execution, through a secure program, to access confidential data. This raises the need to ensure the security of the underlying hardware where the software runs, urging two research problems: First, how can we design secure hardware? Second, given a secure hardware implementation, how can it be verified that it follows the security specification?

In this project, we address both problems in a simpler scenario: we design secure and insecure RSA hardware modules against *hardware timing side-channel* attacks and use *information flow tracking* to verify the security guarantee. A hardware timing side-channel attack assumes the attacker has access to the digital timing (defined as the number of clock cycles) of the computation on the hardware design. If the timing information of the computation is dependent of the confidential data, then the attacker can infer those data through inspecting the timing. Therefore, to

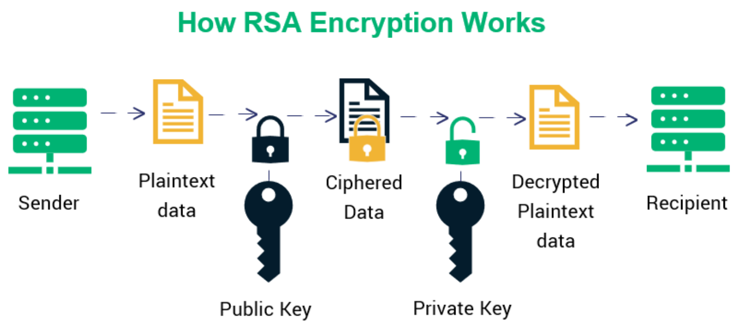

Figure 1: The Overview of RSA (Image reference from www.securew2.com)

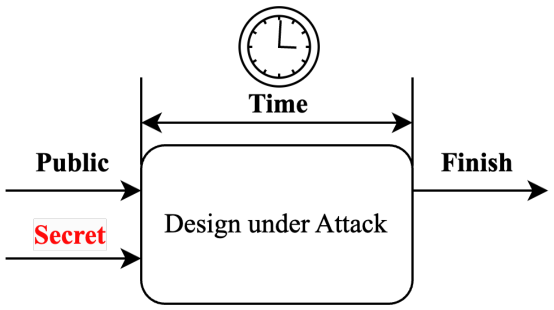

Figure 2: Illustration of how timing side-channel attack is performed on digital hardware design, where the attackers try to infer `Secret` inputs with timing information by manipulating the `Public` inputs and observing the `Finish` signal.

be resistant to such an attack, we require the computation time of the hardware to be independent of the confidential data, often called the *constant time* property.

On the other hand, information flow tracking is a well-established technique used in security verification that tracks the information flow from the source to the destination. In our context, the source is set as the place where the confidential data lie, and the destination is the signals where the attacker can obtain timing information. Checking the security of the hardware under the timing side channel is thus reduced to checking the information flow from the confidential data and timing signals.

We choose the RSA hardware system because it is widely used for encrypting private data and communication. Therefore, it is reasonable to ensure that the execution of such a secure system is free from timing side-channel attacks.

The remainder of this report is structured as follows. Section 2 introduces the background of hardware attacks and hardware security, and also defines the scope of the problem we are interested in in this project. Section 3 explains our implementation of both a naive RSA crypto core and a secure RSA crypto core in Verilog. Section 4 delves into the implementation details of timing side-channel verification methods. Section 5 shows the verification results. Section 6 discusses our observations and outlines the possible future research directions. Last, Section 7 concludes the project. All codes and scripts are available at https://github.com/Chenyungchin/RSA-Timing-Side-Channel.

2 Background

2.1 Information Flow Tracking (IFT)

Information Flow Tracking (IFT) is a security technique used to monitor how information propagates within a system, such as a circuit, providing insights into preventing potential information leakage or unauthorized modification of private data. Figure 3 illustrates an example of IFT on a 2-input AND gate. For this given case, the change of the source node `src` is observable at the destination node `dst`, indicating that information of the upstream source node can be leaked from the downstream destination node. In the context of security verification, we set `src` as signals with

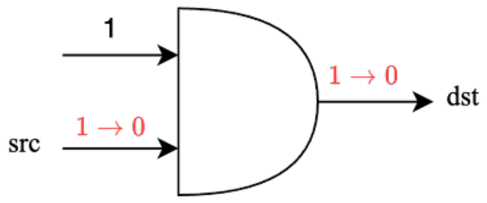

Figure 3: An example of Information Flow Tracking (IFT) on an AND gate. In this case, as the change at `src` from 0 to 1 is observable at the `dst`, there is information flow from `src` to `dst`.

secret values (e.g., the private key d in RSA timing side-channel) and `dst` as signals where the attackers can observe (e.g., the finish time in RSA timing side-channel).

2.2 Problem Statement

In this project, we focus on timing side-channel vulnerabilities in digital RTL modules, where execution time is measured in terms of the number of clock cycles. Given a hardware RTL module with public `finish` output signals and certain private inputs, and assuming unrestricted manipulation of public inputs, our objective is to verify the module's security using IFT based on execution cycles.

3 Implementation of a RSA core

The RSA algorithm begins by generating a public key e and a private key d based on two large prime numbers, p and q. Using the public key e, a plain message m can be encrypted into an encrypted message c. With the private key d, the encrypted message can be decrypted back to the original message m. When p and q are large enough, it becomes computationally impossible for users without the private key d to decode the encrypted data, thus effectively protecting users' privacy.

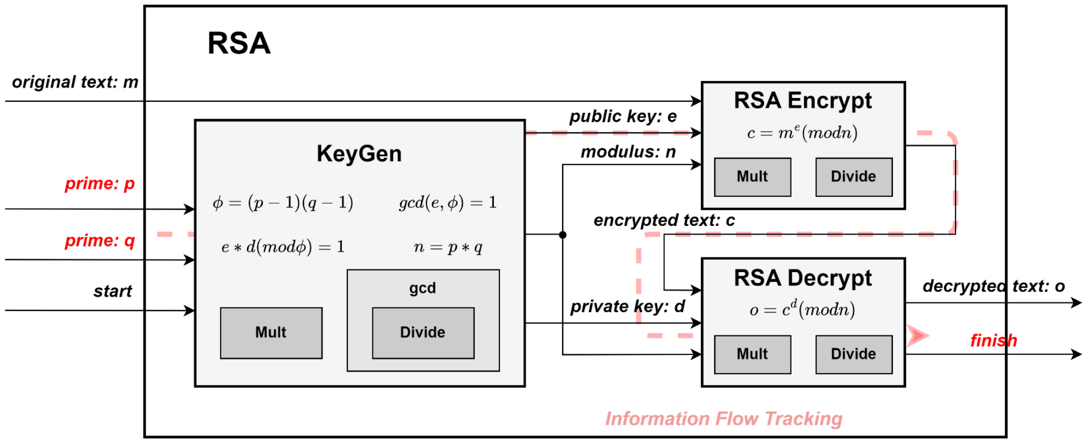

Figure 4: Overview of the RSA module

In this project, the RSA module is modeled as the concatenation of the following three phases: (1) Key Generation, (2) Message Encryption, and (3) Message Decryption, as shown in Fig. 4. Given the two prime numbers p and q, the original message m, and the *start* signal as inputs, the RSA module produces the decrypted message o (which will be the same as the original message m) and the `finish` signal. While typical RSA modules utilize 256-bit or 512-bit precision to ensure data security, an 8-bit case is implemented in this project to facilitate more efficient verification efforts.

3.1 Insecure Implementation

We first implemented a naive RTL module that is susceptible to timing side-channel attack. The implementation details of both the RSA submodules and basic functional blocks are shown below.

3.1.1 Key Generation

The key generation phase computes a key pair e and d and the modulus n given the two prime numbers p and q. This phase implements the following equations: (1) $\phi = (p-1)(q-1)$, (2) $gcd(e, \phi) = 1$, (3) $e \cdot d (\text{mod } \phi) = 1$, (4) $n = p \cdot q$. ϕ. n and ϕ in Equation (1) and (4) are computed using the multipliers. The public key e is initialized to 3 and is updated by $e = e + 2$ until it satisfies Equation (2). The private key d in Equation (3) is calculated using the extended Euclidean algorithm.

3.1.2 Message Encryption

The message encryption phase computes the encrypted message c given the original message m, the public key e, and the modulus n based on the equation $c = m^e (\text{mod } n)$. It is implemented using modular multiplication. The number of operation cycles is positively correlated to the power e and thus is not constant.

3.1.3 Message Decryption

The message decryption phase computes the decrypted message o given the encrypted message c, the private key d, and the modulus n based on the equation $o = c^d (\text{mod } n)$. It is also implemented using modular multiplication with a non-constant number of operation cycles.

3.1.4 Multiplier

The multiplier adopts the traditional shift-and-add fashion, which gives a constant time operation with respect to the bit widths of the inputs.

3.1.5 Divider

A restoring style multi-cycle divider is implemented. Similar to the multiplier, constant-time operation is guaranteed with respect to the bit widths of the inputs.

3.1.6 GCD

The GCD module follows the Euclidean algorithm by iteratively updating the inputs to the divider based on the remainder. The extended Euclidean algorithm is also implemented to compute the private key d. Unlike the multiplier and divider, the operation time of the GCD module varies, as it depends on the number of divisions required until the remainder reaches zero, making it non-constant.

3.2 Secure Implementation

To adapt the RSA module to a secure core resilient to timing side-channel attacks, it is essential to ensure that all submodules on the IFT path have identical operation times, regardless of the secret inputs. This is achieved by analyzing the operations within each submodule and implementing constant-time methods.

3.2.1 Key Generation

In the insecure key generation module mentioned above, two computation steps exhibit non-constant operation times, as shown below.
(1). **The calculation of GCD**: The Euclidean method is an iterative process that terminates when the remainder reaches zero, resulting in execution times that vary based on the secret inputs, making the hardware vulnerable to timing side-channel attacks. To mitigate this, we define a maximum cycle limit, `MAX_GCD_CYCLE`, slightly larger than the worst-case execution time and ensure the `finish` signal is triggered only at the end of this fixed duration. The worst-case scenario occurs when the two inputs to the GCD are consecutive numbers in the Fibonacci sequence, where the worst-case cycle count can be determined from its characteristic equation. Here we set $\texttt{MAX\_GCD\_CYCLE} = 14 \cdot \texttt{WIDTH}^2$, where `WIDTH` is the precision of the prime numbers.
(2). **The search for the public key** e: In our implementation, the public key e is set to be the first number that satisfies $\gcd(e, \phi) = 1$ with e initialized to 3 and updated by $e = e + 2$. In the worst case, e has to be updated by $2^{\texttt{WIDTH-1}}$ times of GCD operation, which results in $\texttt{MAX\_FIND\_E\_CYCLE} = 2^{\texttt{WIDTH}-1} \cdot 14 \cdot \texttt{WIDTH}^2$.

3.2.2 Message Encryption

The execution time for modular multiplication $c = m^e (\text{mod } n)$ is proportional to e, with each iteration consisting of a MULT operation and a DIVIDE operation. As e has the precision of `WIDTH`, we have $\texttt{MAX\_ENCRYPT\_CYCLE} = 2^{\texttt{WIDTH}} \cdot (\texttt{MULT\_CYCLE} + \texttt{DIVIDE\_CYCLE}) = 2^{\texttt{WIDTH}} \cdot 6\texttt{WIDTH}$.

3.2.3 Message Decryption

Similar to message encryption, the execution time for $o = c^d (\text{mod } n)$ is proportional to d. The only difference is that d has the precision of $2 \cdot \texttt{WIDTH}$, resulting in $\texttt{MAX\_DECRYPT\_CYCLE} = 2^{2 \cdot \texttt{WIDTH}} \cdot 6\texttt{WIDTH}$.

We observe that these worst-case cycle counts often exhibit a dependency on `WIDTH` that is superexponential. The simulation and verification of an 8-bit secure RSA core are already time-consuming, highlighting the significant optimization efforts required to implement a constant-time RSA core for commercial 256-bit or 512-bit applications.

4 Verifying the Constant Time Property

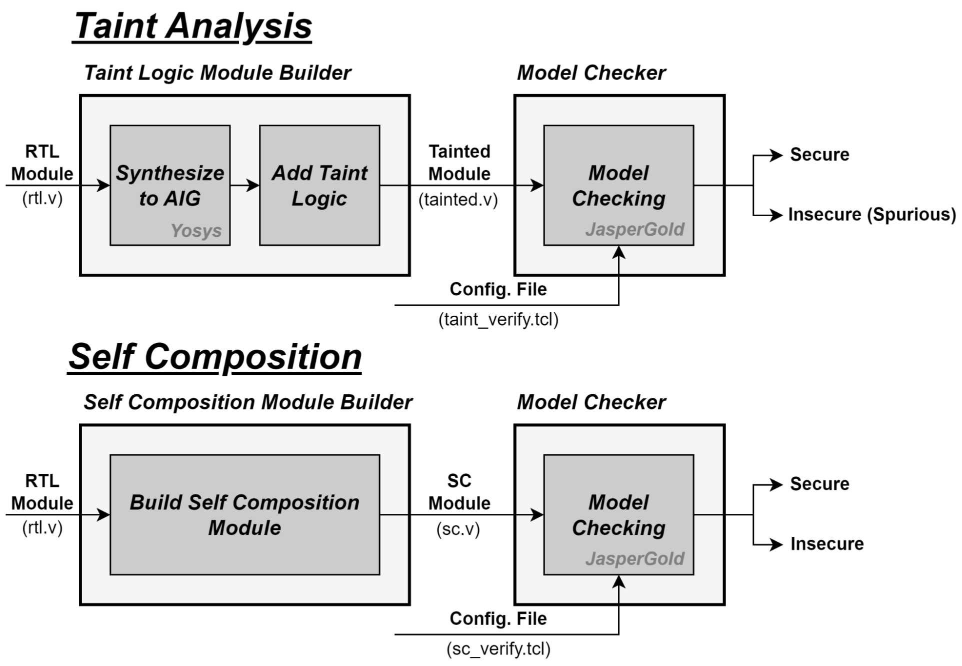

Figure 5: Workflow of the verification procedures for taint analysis and self composition.

We apply **Taint Analysis** and **Self-Composition** to verify the constant time property of the RSA module. The rationales and implementation details of these methods are described below. Notice that our system satisfies constant time property if and only if there is no information flow from the private inputs to the `finish` signal, as indicated by the red dashed arrow in Fig. 4. Therefore, in the following discussion, we focus on how to solve the IFT problem with the two different approaches. Fig. 5 shows the overview of the verification methodology with two different approaches. For taint analysis, we first synthesize the input RTL module (`rtl.v`) into a gate-level netlist with the Yosys [3] synthesis tool and elaborate the gate-level netlist with gate-level taint logic, resulting in the tainted module `tainted.v`. Then the model checking is performed using the commercial model checker JasperGold [4] with a TLC file specifying the parameters of the verification engines, assumptions, and assertions for the constant time property. Similarly, self-composition is done by querying model checking on the self-composition module (`sc.v`) converted from the input RTL (`rtl.v`).

4.1 Verification with Taint Analysis

Taint analysis is a technique that creates a taint signal and a taint propagation rule for each signal within the circuit. The taint propagation rule is designed such that given a signal `s`, its tainted signal `s_t` is HIGH if there exists an information flow from the private signals to s. By definition, the taint signals of the private inputs are always HIGH, and the original statement of checking the information flow from private input to public output can be reduced to checking if the taint

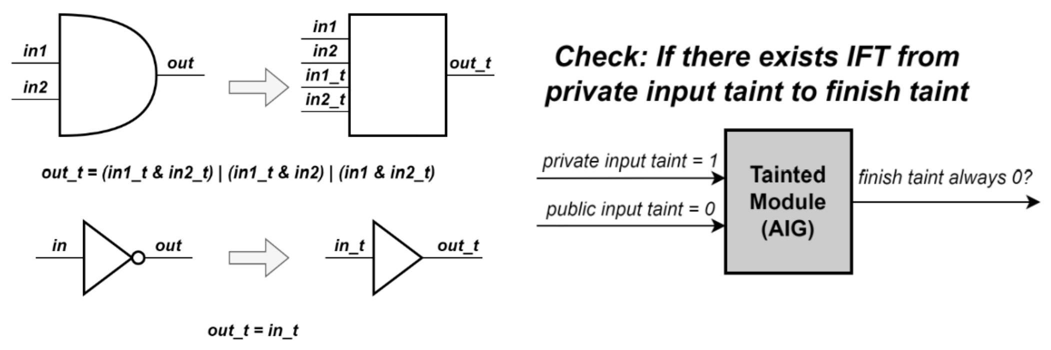

Figure 6: The rationale of taint analysis. The taint can propagate to the next level at a 2-input AND gate if both inputs are tainted or one input is tainted while another input is HIGH. On the other hand, the taint can propagate at an INV gate if the input is tainted.

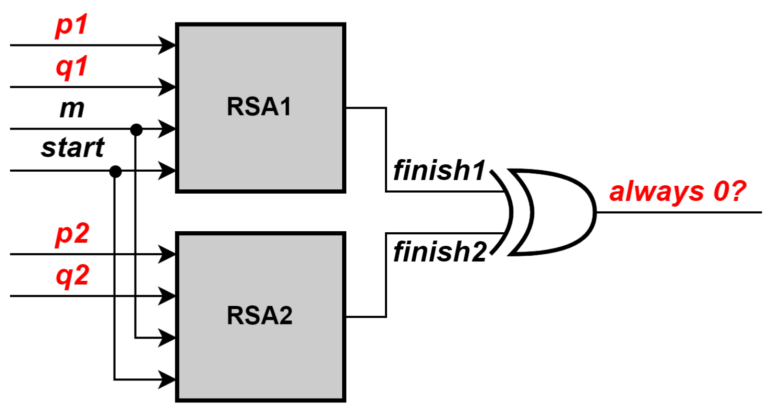

Figure 7: Self composition is done by creating two copies of the RSA modules with common public inputs (`m` and `start`) and different private inputs (`p1`, `q1`, `p2`, and `q2`).

signals of the public outputs can be HIGH. We follow the gate-level taint analysis scheme originally proposed in [5] for creating gate-level taint propagation logic. By defining the taint propagation logic for each primitive logic gate (such as AND, OR, and INV), the taint logic of the output signals can be derived based on the composition of the taint logic of the gates within the netlist.

In our implementation, we first convert the RTL module to and-inverter graph (AIG) and then apply taint logic for 2-input AND gates and INV gates. As shown in Fig. 6, for a 2-input AND gate, the output will be tainted if both inputs are tainted or one input is tainted while another input is HIGH. As for an INV gate, the output will be tainted if the input is tainted. With only these two rules, we can track if an information flow exists from the private input taint to the finish taint, enabling fast and efficient searches for counterexamples. Nevertheless, the defined taint propagation logic only ensures correctness when the inputs are independent. As for the case of dependent input patterns (e.g., a reconvergent path), it is possible to obtain a spurious counterexample. Consequently, an additional check of the counterexample is required.

4.2 Verification with Self-Composition

In contrast to taint analysis, self-composition [6] is a technique that is both sound and complete for proving information flow property, i.e. the result of self-composition is safe if and only if there is no information flow. Self-composition reduced the IFT problem to the classical model checking problem, where the underlying design is the *composition* of the two copies of the module. The composition is done by connecting the public inputs (`m` and `start`) and allowing different private inputs (`p1`, `q1`, `p2`, and `q2`) among the two copies, as shown in Fig. 7. The final assertion is to check whether the public outputs (`finish1` and `finish2`) can be inconsistent. A counterexample for this model-checking query is a trace within two copies with the same sequence of public inputs, different sequences of private inputs, and different public output values at the last cycle. This exactly corresponds to the definition of information flow and the constant-time property. However, since we double the design size by creating two copies, self-composition usually suffers from scalability issues compared to taint analysis.

5 Results

This section analyzes the verification results of **taint analysis**, **self-composition**, and the built-in information flow checker SPV in JasperGold on both the insecure and secure RSA cores under different bit width. We set a 600-second timeout for performing verification on each test case.

Fig. 8 presents the verification times (either to find a counterexample or reach a timeout) for three verification methods across bit-widths ranging from 8 to 16 bits. The results demonstrate that, for a fixed bit-width, taint analysis is often more than 10 times faster than the other two methods. Additionally, taint analysis shows a slower increase in execution time as the bit-width increases, compared to the self-composition method and JasperGold's built-in SPV, highlighting its finer scalability. However, manual validation revealed that all counterexamples produced by taint analysis were spurious, so an iterative blocking mechanism for spurious counterexamples is expected for future work.

For self-composition and JasperGold's built-in SPV, the verification times roughly quadruple when scaling from 8-bit to 10-bit and reach the 600-second timeout for precisions equal to or greater than 12-bit. Both the self-composition method and JasperGold's built-in SPV do not possess great scalability even under a slight bit-width increase of 2. This indicates the difficulty of finding counterexamples of insecure design.

For the secure implementation, all three methods fail to complete the verification within the timeout even with 8-bit bit-width. The results are shown in Fig. 9, where only the number of frames reached is displayed for clarity. We observe that taint analysis reaches fewer frames than the other two methods. Also, the difference in frames reached decreases as the bit-width increases. This suggests that self-composition and SPV may be better for verifying secure designs with small sizes while taint analysis still exhibits slower runtime blows-up according to the trend of the frames reached compared to the other two methods.

On the other hand, when considering both insecure and secure verification, self-composition and JasperGold's SPV exhibit similar behavior in both verification time and frames reached. Therefore, it is likely that JasperGold's SPV uses self-composition-based techniques for checking IFT.

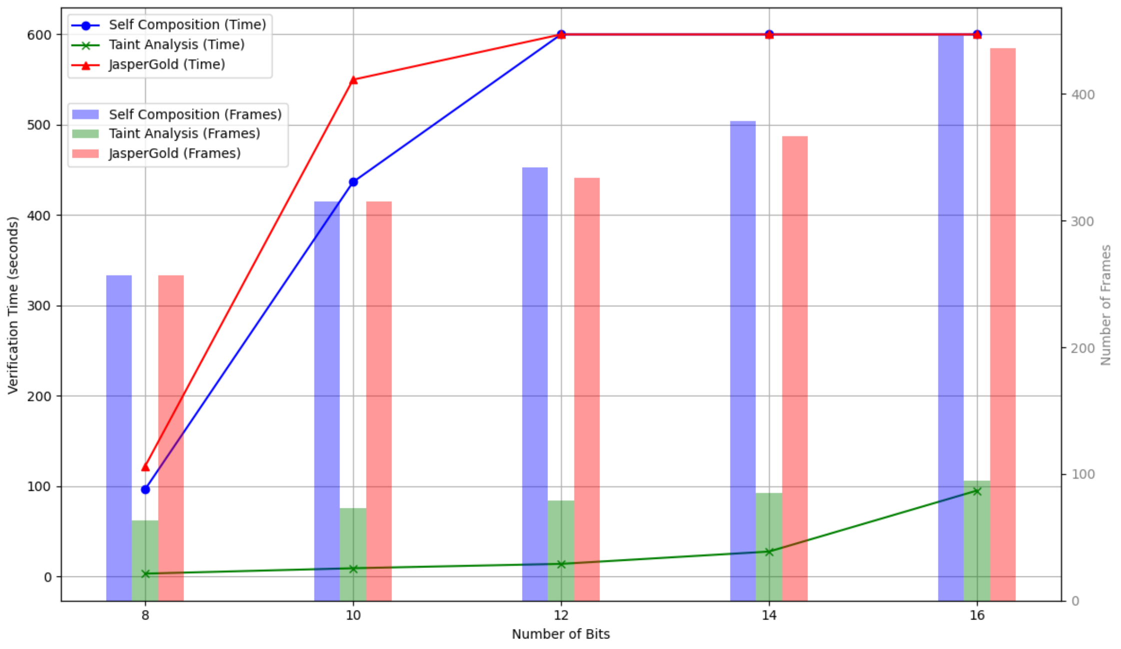

Figure 8: Verification results of insecure implementation with different bit-width. The timeout is set to be 600 seconds.

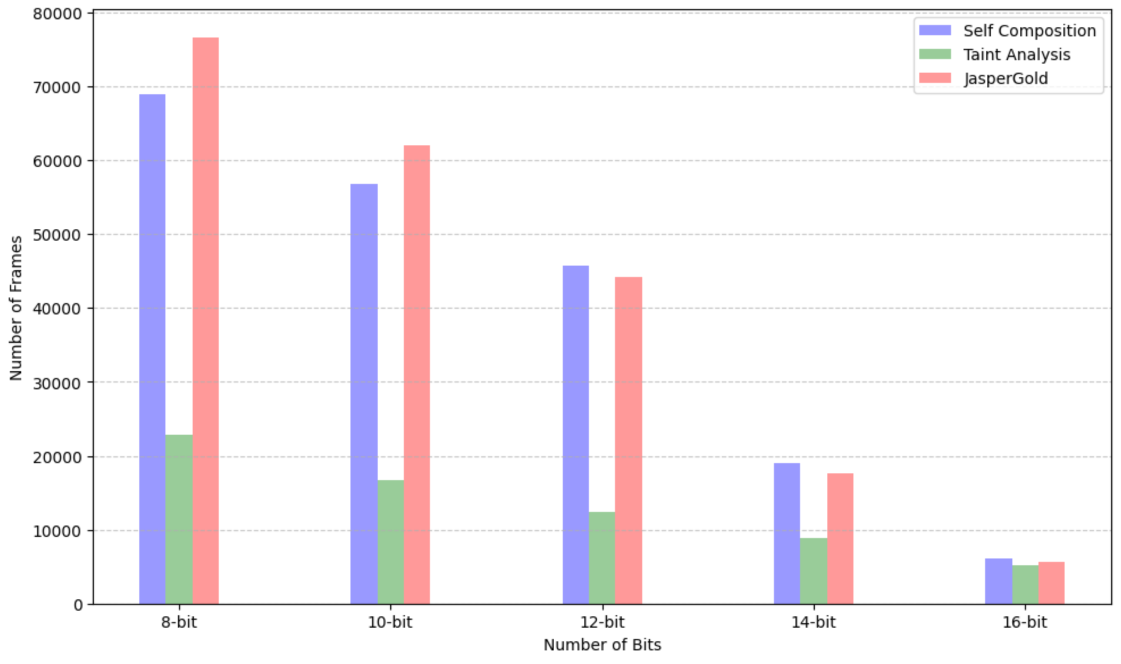

Figure 9: Verification results for secure implementations with different bit-widths are presented. Note that since all three methods failed to complete the verification within the timeout, only the number of frames reached is shown in this figure.

6 Discussion

6.1 Implementation of a Constant-time RSA Core

As illustrated in Section 3, the worst-case execution time of RSA exhibits superexponential correlation to `WIDTH`. For a mere 8-bit implementation, the total number of cycles for performing constant-time key generation, message encryption, and message decryption takes roughly 3000000 cycles. The number of cycles further skyrockets when we scale the precision to a commercial 256-bit/512-bit design, making it infeasible for daily use. Therefore, the hardware implementation for constant-time RSA requires a huge amount of design optimization on each submodule to balance between security and practicality.

6.2 Cone of Influence Analysis for Secure Implementation

In the constant-time implementation, the private inputs do not fall in the cone of influence of the finish signal. As verifying the model's functional independence and constant-time property offers an opportunity to avoid time-consuming exhaustive checks, the constant-time property should be verified syntactically by examining the structural independence between the `finish` signal and the private input. However, based on our experiments presented in Section 5, the model checkers are unaware of such structural information, leading to timeouts for all secure cases. One possible improvement to our current implementation is to include a cone of influence analysis so that we can prove the current secure implementation without traversing through the entire search space exhaustively.

7 Conclusion and Future Work

In this project, we implement both secure and insecure RSA hardware systems against timing-side channel attacks. We apply gate-level taint analysis and self-composition on the RSA design with an off-the-shelf model checker to check the constant-time property. For insecure design, self-composition is able to find a counterexample for a relatively small design size but suffers from scalability issues as the design size increases. While taint analysis has better scalability, the counterexamples it reports are all spurious. For secure design, both methods fail to report either a proof or a counterexample due to the superexponential correlation of the worst-case execution time with respect to the bit-width. For the current secure RSA system, it is observed that the `finish` signal is structurally independent of private inputs, suggesting that the proof can be drawn by simple structural analysis of the circuit/code structure. For future work, we would like to look into more efficient and effective implementation of a constant-time RSA system and see how verification complexity correlates with different implementations. Further, as the current taint analysis can only find spurious counterexamples, we would like to explore ways of blocking/refinement procedures so that the model checker will exclude the spurious cases and search for other traces that may be true counterexamples.

References

[1] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint*

arXiv:1801.01207, 2018.

[2] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[3] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, 2013.

[4] Cadence. Jasper Formal Property Verification App — cadence.com. `https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/formal-property-verification-app.html`. [Accessed 23-06-2024].

[5] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 109–120, 2009.

[6] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.