National Tsing Hua University
Department of Electrical Engineering
EE4292 IC Design Laboratory Fall 2022
Final Project

Digital signature authenticator through SHA256 / SHA3-256 & AES encryption / decryption

電機 23 陳昭維 王瑭毅 林禹陞

1. Functionality

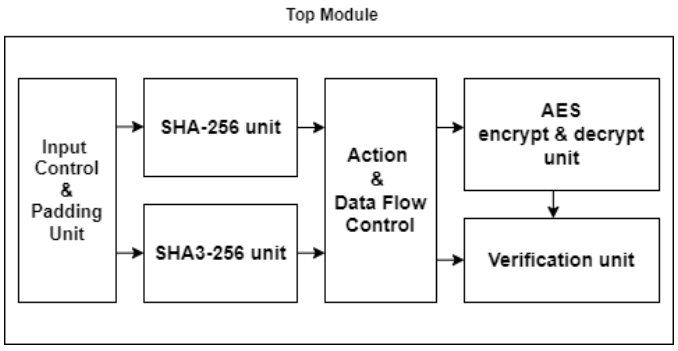

Figure 1 Block Diagram

[Data Flow Overview]:

We aim to implement a digital authentication flow of message or document on hardware. In general, transmitter hashes the message / document through hashing algorithm. Then, encrypted the hashed value with private key with corresponding public key. For the receiver side, they can decrypt the cipher and hash again with the received message / document simultaneously. Eventually, they compared two hashed value, one from decryption, the other from hashing to see whether the message / document is authentic or manipulated.

[Encrypt and Digital Signature Generation]:

In encrypt flow, the message / document is processed by the padding unit, which slices and pads the message into multiple of fixed block size for two hashing units, 512 bits and 1088 bits respectively. The output of SHA256 serves as the key of AES, and the output of SHA3-256 serves as the plaintext of AES when both hashing units finished their iterations. We obtain the ciphertext as the digital signature of this message/document after encryption. Then we can transmit the ciphertext and the original message/document.

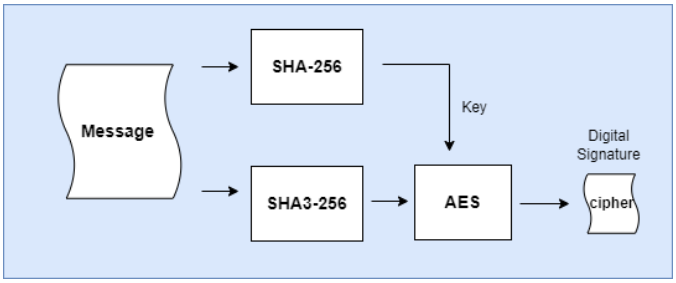

Figure 2 encryption flow

[Decrypt and Verification]:

In decrypt flow, we also process the transmitted message / document into the padding unit and both hashing units to acquire the both hashing value. Since SHA256 module is set to be done before the SHA3-256 module with our design, the output of SHA256, which is the decryption key of the AES module, could start decrypting the transmitted ciphertext. The decrypted ciphertext can then be compared with the hashed message / document in time, where the verification process is done by the verification module.

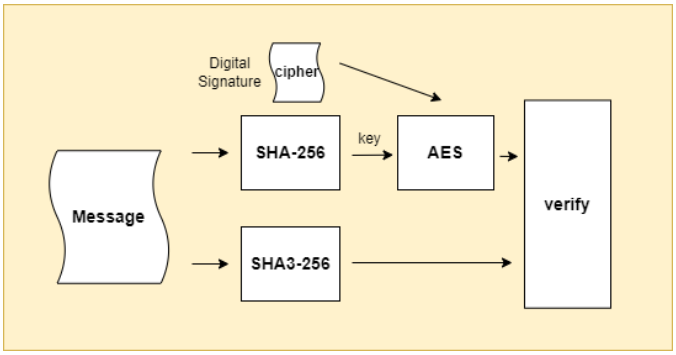

Figure 3 decryption flow

[Novelty]:

Our approach is that we can implement the key generating part from another set of hashing algorithm on the message / document, so that no public key is needed. For the receiver, they first hash the message / document with two given hashing algorithm and then decrypt the cipher then compare the two hashed value. For this protocol, as long as the message or the key is not manipulated, the authentication works and it's very nearly impossible for only manipulating message / document to deceive the receiver with false message / document since the hashing algorithm - SHA256[1] / SHA3-256[2] we applied in this hardware both have low collision rate.

2. Implementation

Type	Name	bits	Description
Input	clk	1	Clock
Input	srst_n	1	System reset
Input	enable	1	Trigger hardware operation
Input	mode	1	encrypt/decrypt
Input	msg_sram_data	64	message data
Input	m_len	14	message length (byte)
Input	cph_sram_data	8	cipher data
Output	valid	1	indicates data validity
Output	verify	1	show that if cipher verifies the message
Output	result	64	for cipher output
Output	msg_sram_addr	11	address for message data
Output	cph_sram_addr	5	address for cipher data

Table 1 I/O definition

[Hardware Design]:

The input control and padding module primarily deals with the padding mechanism of SHA256 and SHA3-256, along with the communication of input data between SHA modules and SRAM for data fetching. It sends the sliced and padded message blocks following the algorithms to SHA256 and SHA3-256 modules. The SHA modules are triggered to perform the hashing algorithm whenever the input message block is available with given load_enable signal for both modules. Moreover, this padding module also controls the valid signals for the two SHA modules if all the iterations of them are already done. These valid signals feed forward to the "Transmission" module, providing the information to the AES module that encryption / decryption can be started.

In terms of saving the SRAM resource, we decided to use just one SRAM to store the message which is being to be processed. This decision enormously increases the complexity of this padding module because the iteration cycles of SHA256 and SHA3-256 is totally different, and we can only fetch the input data from SRAM one time at a cycle. Furthermore, since we set the data input port to be 64-bit wide, we cannot fetch one complete message block in one cycle. But the solution is, we take the advantage of the SHA algorithm that they must do the iteration of at least 64 cycles for every message block, combined with the usage of the "busy signal" and the control of FSM, we made the padding module pre-prepare the next message block for both SHA modules in advance according to their processing speed, enabling them to do the hashing operation back-to-back for every message block without any additional cycle for data fetching, achieving the performance of the behavior of using two SRAMs.

Padding mechanism for both hash units are the other complicated part of tackling the input message. Since for simply SHA256 and SHA3-256, there are already 5 different situations we need to consider for padding.

Figure 4 Padding Rules for SHAs

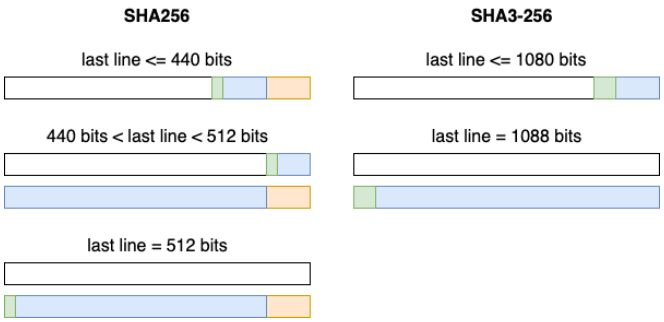

Figure 5 Padding Scenario

We calculate the remain length of message for every message block and every 64 bits in advance, respectively, combined with deciding whether the next input is to be padded or not before fetching the next 64-bit input, or the padding is done. When the remaining length is calculated to be zero, the valid signal of SHA256 or SHA3-256 is set to high when the final result is done.

We also set the maximum input message length to be parameterized, hence if we need to increase the maximum input message length, we can simply adjust the bit width of message length. We set the maximum input length to be $2^{14} = 16384$ (bytes) for the concern of simulation time not being too long.

After the whole design are merged, we found that the critical path is inside the SHA256 module. However, rather than pipelining on the SHA256 module, we turned to optimize the SHA3-256 module for lower cycle count. We did this because the pipelining on the SHA256 module led to two times the number of cycles owing to the iteration algorithm. In addition, since the message block size of SHA256 is nearly half of SHA3-256, the cycles increased relative to SHA3-256 module are $2 \times 2 = 4$ times of the origin, which leads to the delay of output of SHA256, even slower than the output of SHA3-256, violating the setting of our design. In other words, we hope that the output of SHA256 is done before SHA3-256, then taking the output of SHA256 as the key of AES. After doing the decryption, it will be compared with the SHA3-256 output in time. Hence, we decided to reduce the total cycle count of SHA3-256 module, and further increase the throughput of our design.

The SHA3-256 algorithm consists of 24 rounds of 5 consecutive bit permutation functions (*Theta, Rho, PI, KAI, IOTA*). We originally implemented each round in 7 cycles (3 cycles in Theta function) to optimize timing constrain. After merging designs of all modules, the critical path is revealed to be at the SHA256 module and it's not worthy of cutting it down. Thus, we reduce the cycle number for each round in SHA3 algorithm to increase the throughput of hashing.

For the AES part we implemented through the reference algorithm [3]. It's a symmetrical encryption algorithm that ones can decrypt the cipher using the same encryption key. For most of the code we implemented on our own but for the Galois field operation of *inverse mixing row* in AES algorithm, we referenced and compared the correctness through the work of author Joachim Strömbergson [5].

To support the encrypt mode and decrypt mode, we implement some logic to control the data flow depending on the mode. Moreover, considering the data are 256-bit, this transmission unit should consist of flip-flop to ensure the response of the circuit would not too slow. And as it receives the done signal of SHA256 and SHA3-256, it would trigger the AES module working.

After AES module completes encryption or decryption, the verification module would send out output information based on the mode. the encrypting result in 256 bits in encryption flow, or the validity after comparing the decrypting result with SHA3-256 hashing value in a 1/0 encoded signal in decryption flow.

3. Verification

First of all, we have implemented the algorithm of SHA256, SHA3-256 and AES (ECB mode) by Python. And we have check out that the correctness of the Python code by comparing the results with verified generators of SHA256, SHA3-256 and AES

on the Internet. With the help of these Python codes, we can generate the test pattern and golden data. These test patterns contain the mentioned 5 specific conditions to ensure our RTL model can work successfully for any length of message.

Our module support two operating mode, encryption mode and decryption mode. For encryption mode, it needs the content of message, and the length of the message as input, and it would give us the digital signature of the message as output. To be mentioned, the result is fixed 256 bits. To provide the digital signature, our module sets the valid as "1" for 4 cycles. From MSB to LSB, our model expresses 64 bits for each cycle. After collecting 256-bit digital signature, we will compare with the golden data to ensure the result of our module is correct.

On the other hand, for decryption mode, it needs the content of the message, the length of the message, and a digital signature as input, and it would tell that if the provided digital signature belongs to the message. Then we will check out the verification of our module is equal to the golden data to assure that our module can be able to identify the authenticity of the given message / document and digital signature.

4. Specification

[Throughput – Encryption Flow]:

For our design, there is always a fixed number of cycles for fetching first block of data and the time spent on AES encryption. For short length messages, the throughput of our design will be significantly low due the cost of the operations mentioned. As the message gets longer, the time spent on hashing will dominates the whole flow which dilutes the cost of the fetching and encrypting time. Also, the padding mechanism of our design doesn't require additional cycle to prepare the next message block, the throughput increases. As the message length increases, the throughput gradually converges to a value, which is approximately 290Mbyte per second.

The throughput is approximated by the following formula,

$$\frac{n \times SHA3\ block\ size}{(n \times cycles\ of\ SHA3 + cycles\ of\ AES) \times cycle\ time}$$

As n increases, throughput is dominated by SHA3-256 operation,

$$\frac{SHA3\ block\ size}{cycles\ of\ SHA3 \times cycle\ time} \approx 290\ (Mbyte/s)$$

	pattern						
	#1	#2	#3	#4	#5	#6	#7
message length (byte)	3	59	64	136	1253	1857	14459
cycles	230	230	230	375	1535	2116	15600
throughput (Mbyte/s)	4.08	80.16	86.96	113.33	255.09	274.25	289.64

Table 2 Throughput vs. length of message

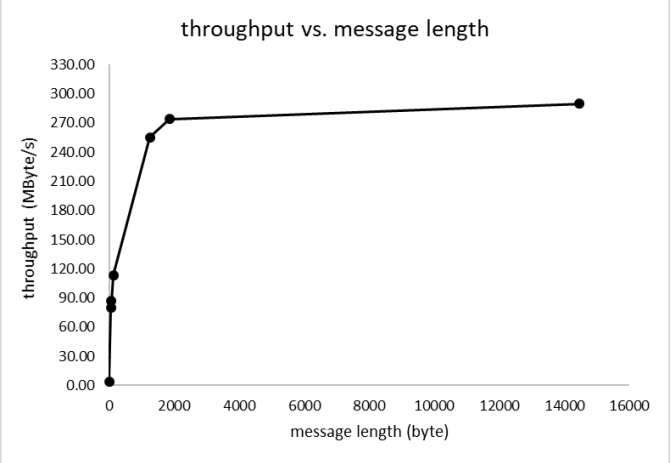

Figure 6 Throughput vs. message length

[Timing / Area / Power Analysis]:

	timing (ns)
synthesis timing	2.35
APR timing	3.2

Table 3 Timing for synthesis and APR

	area (um^2)	utilization
synthesis area	241509.46	85%
APR area	284199.26	

Table 4 Area of synthesis and APR

Power Analysis	Pre Layout	Post Layout (pre-sim waveform)	Post Layout (post-sim waveform)
Net Switching Power (W)	2.76E-04	1.30E-03	1.38E-03
Cell Internal Power (W)	2.06E-03	2.12E-03	2.22E-03
Cell Leakage Power (W)	6.03E-06	7.31E-06	7.31E-06
X Transition Power (W)	2.65E-07	5.54E-07	5.02E-07
Glitching Power (W)	7.61E-07	1.23E-04	1.02E-06
Total Power (W)	2.35E-03	3.43E-03	3.60E-03

Table 5 Power analysis

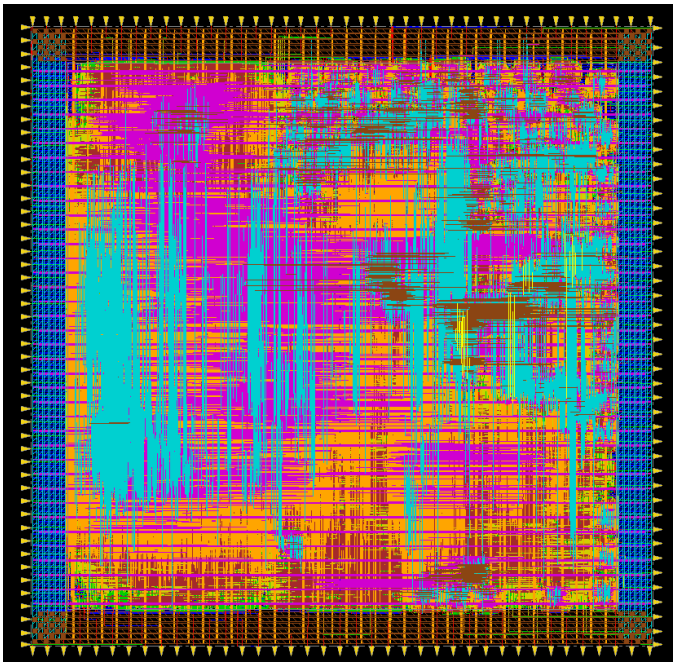

Figure 7 APR Physical View

5. Conclusion

In this project, we first targeted to design a hardware that can support multiple hashing level, but the hardware resource sharing stood as a difficulty due to the heterogeneity of different hashing algorithms, so we instead implemented a special flow of digital authenticationon hardware through SHA256 and SHA3-256 algorithm combined with AES encrypt / decrypt algorithm. We also applied some new idea on the digital authentication flow such as the key generation method with some timing concern in decryption. Finally, we did APR with core utilization of 85% for area concern, which is higher than a normal chip's performance, and with a acceptable timing and throughput.

6. Reference (Documentation and Source code)

[1] Secure Hash Standard., Federal Information Processing Standards Publication., August 1 2002.

[2] Penny Pritzker, Willie May., SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions., Federal Information Processing Standards Publication. August 2015.

[3] Advanced Encryption Standard., Federal Information Processing Standards Publication., November 26, 2001.

[4] github: secworks/sha256/src/model/sha256.py
Author: Joachim Strömbergson
Copyright © 2013 Secworks Sweden AB
All rights reserved.

[5] github: secworks/aes/src/rtl/
aes_decipher_block.v.
Author: Joachim Strömbergson
Copyright © 2013, 2014, Secworks Sweden AB
All rights reserved.

7. Contribution

[Software]

SHA3-verification python code: 林禹陞、陳昭維

Data flow verification code: 王瑭毅

Pattern/Golden generation code: 王瑭毅

[RTL]

Top module: every member contributed

Input data and padding control: 林禹陞

SHA256 unit: 林禹陞

SHA3-256 unit: 陳昭維

AES unit (exclude the gf operation in decrypt mode): 陳昭維

Verification module: 王瑭毅

Testbench for pre/gate/post simulation: 王瑭毅

[Presim + Gatesim & APR & Postsim]

Every member contributed and verified

[Documentation and File/Source Organization]

Every member contributed

[Presentation]

Powerpoint: Every member contributed

Interim presentation : 陳昭維

Final presentation: 王瑭毅