TEE OS : Side Channel Resistant Crypto Library for TEE

Sun Yongkang
11911409

Wang Chenyu
11911104

I. Abstract

To protect the running environment of security-sensitive programs in computing devices, researchers proposed TEE technology, which provides a safe running environment for security-sensitive programs isolated from the general computing environment by isolating hardware and software. Side-channel attacks are based on information obtained from the physical implementation of a cryptosystem. For example, time information, power consumption, electromagnetic leakage, or even sound can provide additional sources of information that can be used to further hack the system. The TEE architecture only provides an isolation mechanism but can not resist this type of emerging software side-channel attacks. In this project, we purpose a side-channel attack resistible crypto library by pruning and modifying an existing open-source one. Our library will contain the most commonly used crypto algorithm used in an OS (ECDSA, RSA, etc.).

II. Motivation and Introduction

A. Research Purpose

As a part of the TEE-OS group, our final purpose is to build a complete secured operating system in *TEE* (Trusted Execution Environment). We break it into several parts. And our group is responsible for creating a *side channel resistant crypto library* with *RUST* programming language, used in our final TEE-OS.

B. Research Significance

a) Why need an OS in TEE: Firstly, "*trusted execution environment (TEE)* is a secure area of the main processor. It guarantees code and data loaded inside to be protected concerning confidentiality and integrity. A TEE as an isolated execution environment provides security features such as isolated execution, the integrity of applications executing with the TEE, along with confidentiality of their assets." according to Wikipedia (1). In a conclusion, TEE can promise us:

- *Data Confidentiality*
- *Data Integrity*
- *Code Integrity*

Secondly, to run applications in TEE, we have several ways as shown in figure 1. It is a trade-off between security and usability. Besides these three methods, we plan to implement a mini-OS inside TEE. Through this way, we can have a balance between security and usability.

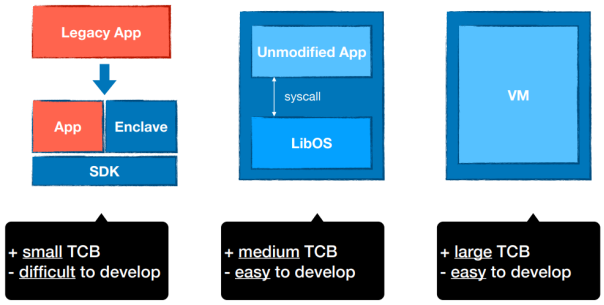

Fig. 1: TEE Development Model

b) Why Rust: Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety. *Rust* can also ensure memory safety through features like Ownership and Borrowing.

c) Why OS need a crypto library: Cryptography can be used for the following purposes:

- Confidentiality: Prevents the user's identity or data from being read.
- Data integrity: Preventing data from being changed.
- Authentication: Ensuring that data is sent from a particular party.

For example, an OS in TEE needs to encrypt its data in the cache to store it in memory, and also some TEE has the ability to do authentication remotely which need asymmetric cryptographic algorithm like *RSA*.

d) Why need side channel attack resistance: Many widely used encryption algorithms have been hacked through side-channel attacks (SCA). For example:

- Osvik, Shamir, Tromer, 2006: Recover AES-256 secret key of Linux's dmcrypt in just 65 ms, according to Osvik, Shamir and Tromer, 2006 (2)
- AlFardan, Paterson, 2013: "Lucky13" recovers plaintext of CBC-mode encryption in pretty much all TLS implementations, accoeding to AlFardan and Paterson, 2013(3)
- Yarom, Falkner, 2014: Attack against RSA-2048 in GnuPG 1.4.13: "On average, the attack is able to recover 96.7% of the bits of the secret key by observing a single signature or decryption round.", accoeding to Yarom and Falkner, 2014(4)
- Benger, van de Pol, Smart, Yarom, 2014: "reasonable level of success in recovering the secret key" for OpenSSL ECDSA using secp256k1 "with as little as 200 signatures", accoeding to Benger, van de Pol, Smart and Yarom, 2014 (5)

Also, although TEE is well secured, it can do nothing to avoid SCA unless the programmer modifies it at the source code level.

III. Research Approach

A. research method

Our research is based on three steps:

a) First Step: To begin our research, we have to choose and learn to use a current existing TEE, to test our crypto algorithms inside it.

As a mature and most widely used choice, we choose *Intel SGX*. "Intel's Software Guard Extensions (SGX) is a set of extensions to the Intel architecture that aims to provide integrity and confidentiality guarantees to security-sensitive computation performed on a computer where all the privileged software (kernel, hypervisor, etc.) is potentially malicious." according to an article of a general description of SGX (6). Below in figure 2 is the basic model of SGX. Further description of SGX can be found in the article (6).

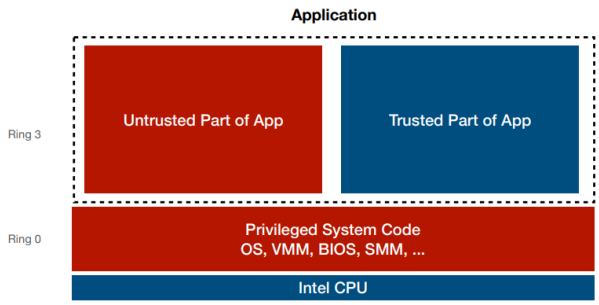

Fig. 2: SGX

Further, We choose to use a SGX development tool, named *TEACLAVE*, which is build up with three layers:

- At the bottom is the Intel SGX SDK implemented using C/C++ and assembly.
- The middle layer is Rust's FFI (Foreign Function Interfaces) to C/C++.
- At the top is the Teaclave SGX SDK.

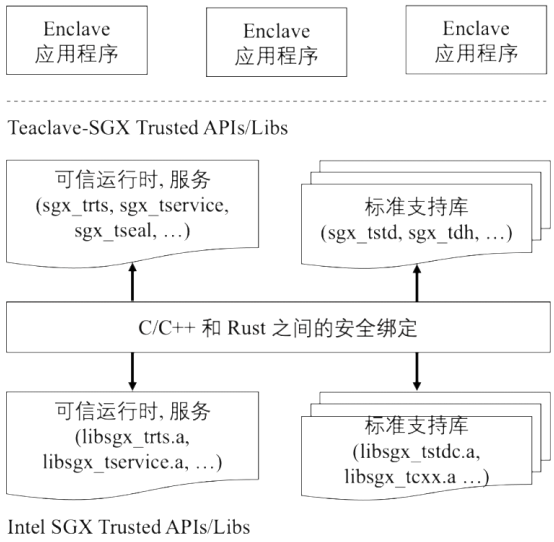

Fig. 3: teaclave

By using these tools, we could develop based on only the topmost Teaclave SGX SDK. Further description can be found on TEACLAVE's website (7)

b) Second Step: Read and learn the current existing Rust crypto library (8), which is an open-source project that contains most of the cryptographic algorithms written in pure Rust.

This library contains algorithms like *AEADs*, *hashes*, *RSA* and etc. We will prune some of these algorithms and test them to use them in Intel-SGX.

c) Third Step: Our goal is to modify the existing crypto algorithms to make side-channel attacks resist. So, our third step is to apply a different kind of side-channel attack to the original code. And find the weak point of the code against our attack.

A list of attacks can be conducted to RSA, DSA, and Diffie-Hellman Key Exchange is concluded by an article by "Bundesamt für Sicherheit in der Informationstechnik", 2013 (9), which gives an overview of relevant literature about side-channel attacks on implementations of either integer factorization cryptography or discrete logarithm cryptography.

More attacks about ECC can be found in an article by "Federal Office for Information Security", 2016 (10).

This document provides a guideline for security evaluators to test implementations of elliptic-curve cryptography over Fp for resistance against side-channel attacks with high attack potential according to version 3.1 of the Common Criteria (CC).

d) Fourth Step: After applying attacks to the original code in the library, we have to modify the code against these attacks based on the data obtained. The above two articles not only contain different attacks but also include methods against each attack. So, our fourth step is to study the article and modify our code according to it.

After being modified, we plan to attack it again to check the effectiveness of our defending.

B. research content

Since learning and modifying a new algorithms takes up a great time, so our team focus on mainly two signature algorithms:

- RSA
- ECDSA

a) Stage One: According to our research methods, our first step is to apply *Intel SGX* and *Teaclave*. And currently, we have finished these parts successfully. We build up the *incubator-teaclave-sgx-sdk* on an Ubuntu virtual machine and use it to simulate an SGX environment. Then we run the basic demos stored in the SGX-SDK described above. Those demos are using RUST language and C language.

Fig. 4: teaclave demo

Fig. 5: teaclave running c code

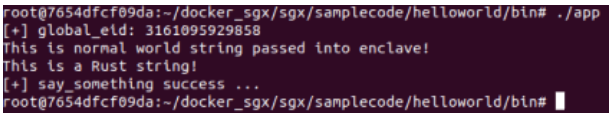

Fig. 6: result

b) Stage Two: Our second step is to learn basic knowledge of our target crypto algorithms *RSA* and *ECDSA*. Since we are a group of two people, so each one of us mainly focuses on only one algorithm. We spend about a week learning basic mathematical background knowledge and the signature process of it. Furthermore, we have done an introduction on our weekly group report to share this knowledge.

- RSA : The RSA digital signature algorithm is based on the RSA public-key cryptography algorithm and uses the RSA public-key cryptography as a digital signature algorithm. RSA digital signature algorithm is the most widely used digital signature algorithm so far. The implementation of the RSA digital signature algorithm is the same as that of the RSA encryption algorithm.

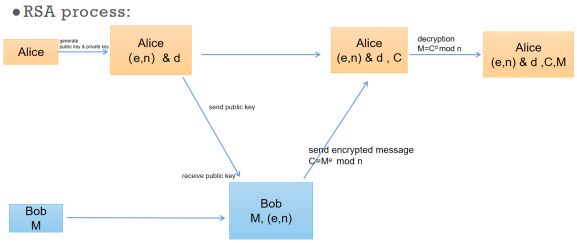

Fig. 7: RSA process

  - Mathematical Background : *Fermat's Little Theorem* and *Euler's Theorem*.

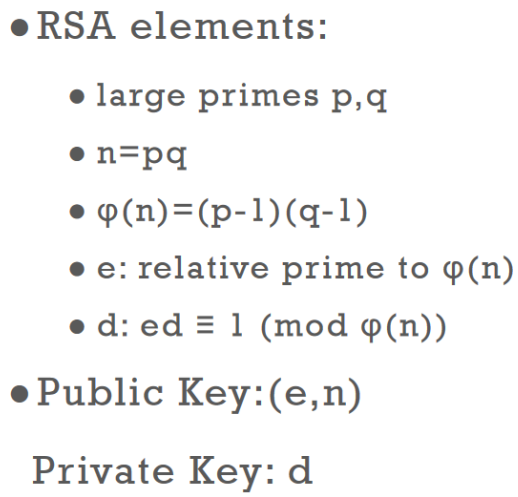

Fig. 8: RSA elements

  - The signature process :
    * 1) The message sender generates a key pair (private key + public key) and sends the public key to the message receiver.
    * 2) Message sender uses a message-digest algorithm to encrypt the original text (the encrypted ciphertext is called digest).

* 3) The sender uses the private key to encrypt the above digest into ciphertext – this process is called signature processing, and the resulting ciphertext is called a signature (note that the signature is a noun).
* 4) The message sender sends the original text and ciphertext to the message receiver.
* 5) The message receiver uses the public key to decrypt the ciphertext (that is, the signature) and get the digest value content1.
* 6) The message receiver uses the same message-digest algorithm as the message sender to encrypt the original text and get the digest value content2.
* 7) Compare if content1 is equal to Content2. If it is equal, the message has not been tampered with (message integrity) and the message came from the sender above (since no one else can forge the signature, this completes ”deniability” and ”authentication of the source”).

- RSA Ecryption (M: message to send):

  $C = M^e \bmod n$

- RSA Decryption:

  $M = C^d \bmod n$

Fig. 9: RSA

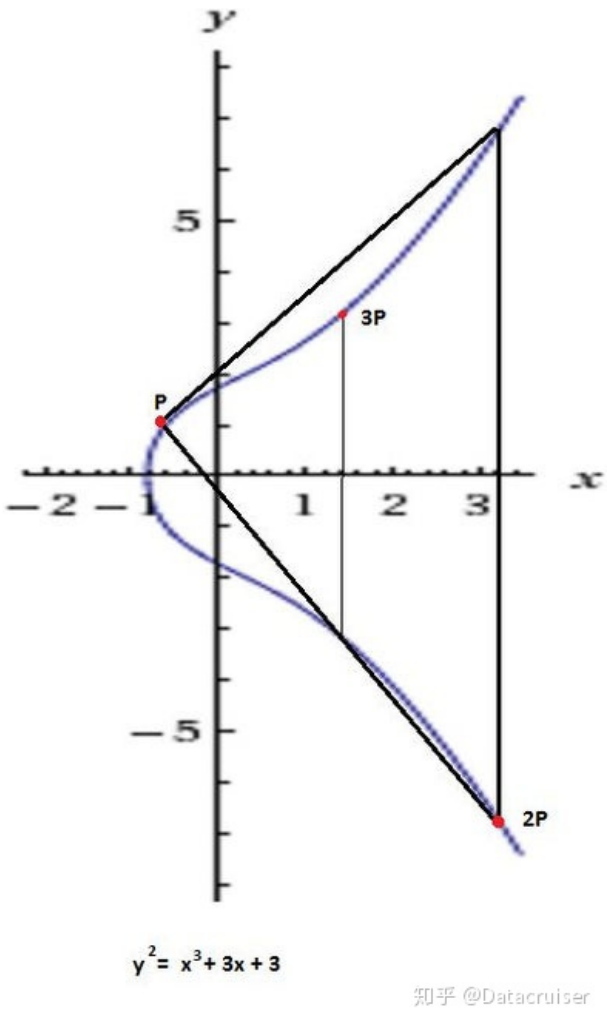

Fig. 10: elliptic curve

- ECDSA : Elliptic Curve Digital Signature Algorithm (ECDSA) is a simulation of a digital signature algorithm (DSA) using elliptic curve cryptography (ECC). The security of elliptic curve cryptosystems is based on the difficulty of the elliptic curve discrete logarithm problem (ECDLP). Elliptic curve discrete logarithm problem is much more difficult than discrete logarithm problem, the unit bit strength of elliptic curve cryptosystem is much higher than that of traditional discrete logarithm system.
  - Mathematical Background : *basic knowledge of elliptic curve*.
  - The signature process :
    * The signature process is as follows:
      · 1. Select an elliptic curve, Ep(a,b), and base point G;
      · 2. Select the private key k (k¡n, where n is the order of G), and calculate the public key K = k * G using base point G;
      · 3, generate a random integer r (r¡n), calculation point r = r * G;
      · 4. Take the original data and the coordinate values of point R x and y as parameters, calculate SHA1 as hash
      · 5, Computing S r - Hash * k (mod n)
      · 6. R and S are the signature values. If either r or S is 0, perform step 3 again
    * The verification process is as follows:
      · 1. After receiving the message (m) and signature value (r, S), the receiving party performs the following operations
      · 2, calculation: s * G + H(m) * P = (x1,y1), R1 x1 mod P

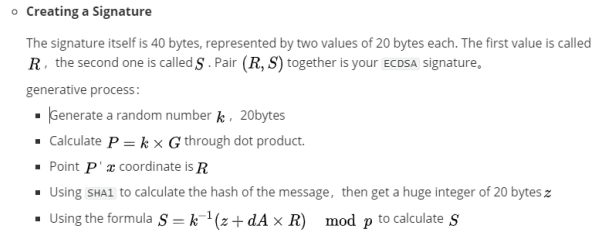

Fig. 11: signature process

- 3, verify the equation: R1 r mod p
- 4, if the equation is true, accept the signature, otherwise the signature is invalid.

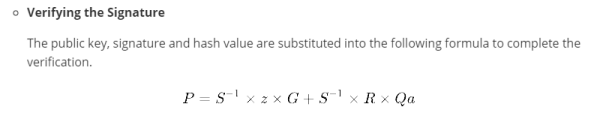

Fig. 12: verification process

Then we use nearly one week to read the source code and prune some of the code we do not need. For example, below is the basic code demo and project structure of *ecdsa* algorithms.

c) Stage Three: The third step requires us to run the pruned code inside SGX and then do some side-channel resistant tests toward these codes. We are still inside this stage. We have now finished pruning some of the code in the rust crypto library, and still trying to compile and use it inside SGX. We currently face some of the compiler version problems. We believe we could solve it soon. However, we have begun to learn some basic knowledge of side-channel attacks, and find some tools to test if a code has side-channel resistivity.

- - https://github.com/agl/ctgrind
- - https://github.com/simple-crypto/SCALib
- - https://github.com/s3team/Abacus

d) Stage Four: We have not reached this stage yet, and we are planing to begin learning and modifying the crypto code soon.

IV. Future Plan

Because of the current schedule, we plan to finish code prune in the following week and test it inside the Intel SGX environment.

And then we will spend about a week learning the basic knowledge of Side-Channel attacks and the testing tools mentioned above.

After that, we should implement the test toward our pruned crypto library and find the defect.

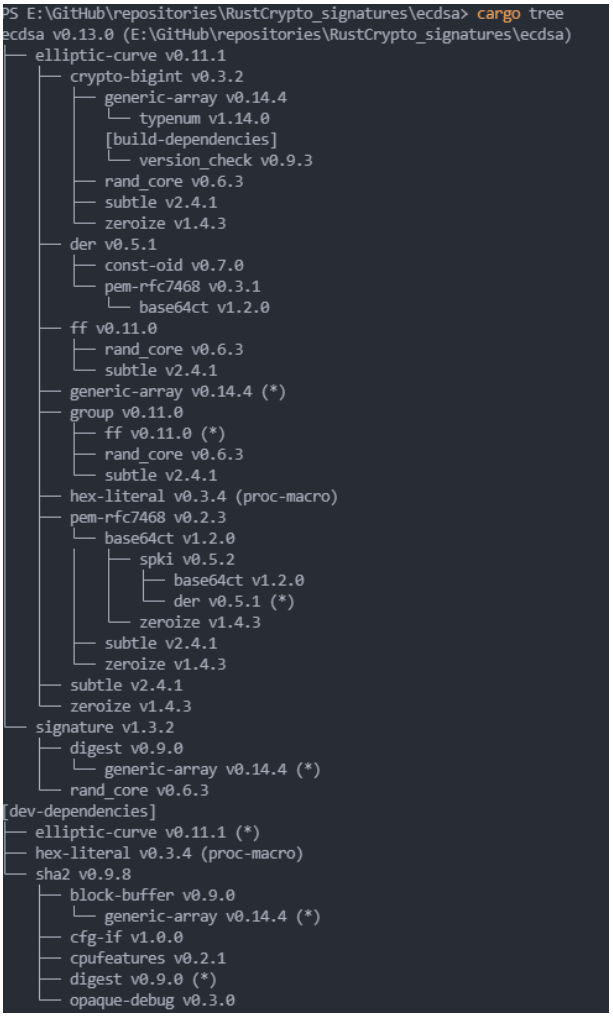

Fig. 13: project structure of ecdsa

Finally, solve those defects according to some related works which are already mentioned in the proposal report. Also, we will test our modified code, and comparing the previous data.

V. Timeline

- Before Week 6: Learn some basic concepts of TEE and Rust programming language. Also, we learned basic usage of Intel SGX, and successfully build up the environment our research needs. *(Finished)*
- Before Week 9: Finish learning existing Rust crypto library. Prune and try to use this existing library inside TEE (Intel SGX). *(Finished)*
- Before Week 12: Finish learning background knowledge of Side-Channel Attacks. Trying to find ways to perform side-channel attacks on these existing libraries. *(Still in progress)*

- Until Week 13: Finish learning methods to resist side-channel attacks, and try to modify the most commonly used two algorithms: RSA and ECDSA.
- Until Week 14: Test and Debug the ability of RSA and ECDSA resisting SAC.
- Until Week 15: Complete other algorithms. Test the performance and ability of our modified library.

REFERENCES

[1] trusted execution environment - wikipedia$_2$021 (2021).
URL https://en.wikipedia.org/wiki/Trusted_execution_environment

[2] E. Tromer, D. A. Osvik, A. Shamir, Efficient cache attacks on aes, and countermeasures, Journal of Cryptology 23 (1) (2009) 37–71. doi:10.1007/s00145-009-9049-y.

[3] N. J. AlFardan, K. G. Paterson, Lucky thirteen: Breaking the tls and dtls record protocols.
URL http://www.isg.rhul.ac.uk/tls/Lucky13.html#Team

[4] Yarom, Falkner, Flush + reload: a high resolution, low noise, l3 cache side-channel attack.
URL http://eprint.iacr.org/2013/448/

[5] Benger, van de Pol, Smart, Yarom, "ooh aah... just a little bit": A small amount of side channel can go a long way.
URL http://eprint.iacr.org/2014/161/

[6] V. Costan, S. Devadas, Intel sgx explained.
URL https://eprint.iacr.org/2016/086.pdf

[7] blog of tool teaclave (2021).
URL https://teaclave.apache.org/blog/2021-08-25-developing-sgx-application-with-teaclave-sgx-sdk/

[8] Rust crypto library (2021).
URL https://github.com/RustCrypto

[9] Minimum requirements for evaluating side-channel attack resistance of rsa, dsa and diffie-hellman key exchange implementations (2013).
URL https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_46_BSI_guidelines_SCA_RSA_V1_0_e_pdf.pdf?__blob=publicationFile&v=1

[10] Minimum requirements for evaluating side-channel attack resistance of elliptic curve implementations (2016).
URL https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_46_ECCGuide_e_pdf.pdf?__blob=publicationFile&v=1