Side Channel Resistant Crypto Library for TEE

Sun Yongkang
11911409

Zhang Kunlong
11911716

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 crypto system. 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. 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. Our research also aims to produce a new way of testing side-channel vulnerability in the source code level of rust, which is dependent on two different testing tools - ‘MIRAI‘ and ‘DATA‘.

B. Research Significance

1) 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.”. 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.

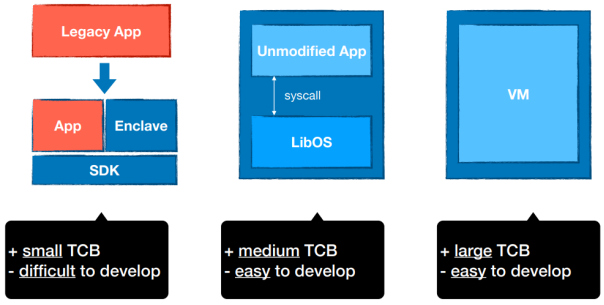

Fig. 1. TEE Development Model

2) Why Rust: Rust is a systems programming language with excellent execution speed, prevention of segfaults, and guarantee of thread safety. *Rust* can also ensure memory safety through features like Ownership and Borrowing.

3) 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*.

4) 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 [17]
- AlFardan, Paterson, 2013: "Lucky13" recovers plaintext of CBC-mode encryption in pretty much all TLS implementations, accoeding to AlFardan and Paterson, 2013 [8]
- 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 [22]
- 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 [9]

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

III. Background

In this section, we will introduce some concepts we will use in our report. Since learning and modifying new algorithms takes up a great time, so our team focus on mainly two signature algorithms:

- RSA
- ECDSA

In the introductions, we will talk about some basic concepts of different encryption algorithms. Besides, we will also talk about some basic knowledge of the Side-Channel Attack and the choosing TEE - SGX with its corresponding simulator - "Teaclave".

A. *TEE - SGX!*

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 [11]. Below in figure 2 is the basic model of SGX. Further description of SGX can be found in the article [11].

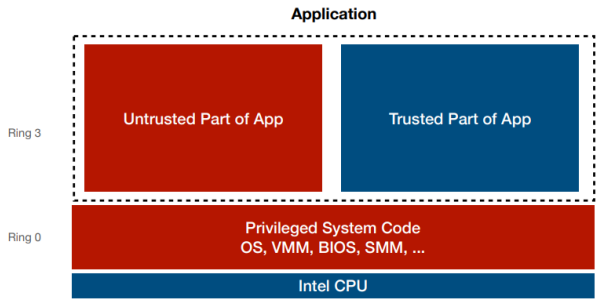

Fig. 2. SGX

B. *Teaclave!*

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.

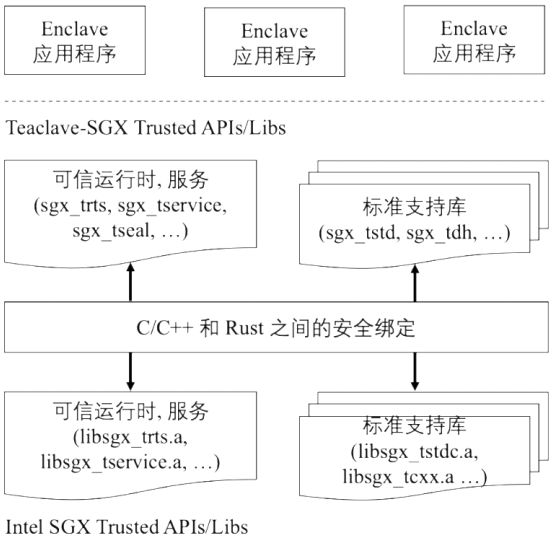

Fig. 3. teaclave

By using these tools, we could develop based on only the topmost Teaclave SGX SDK. Further description can be found on TEACLAVE's website [2]

C. A Crypto Algorithms - 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

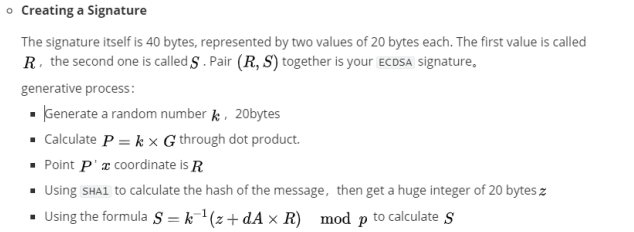

Fig. 4. signature process

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

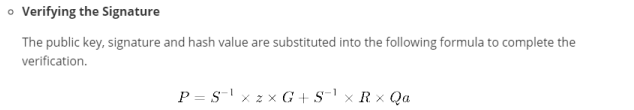

Fig. 5. verification process

D. A Crypto Algorithms - 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.

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

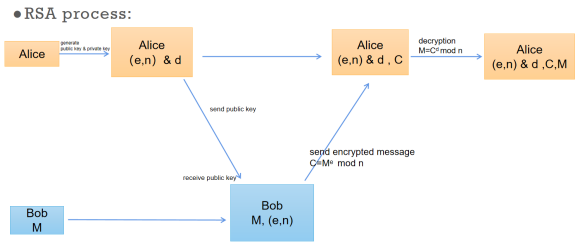

Fig. 6. RSA process

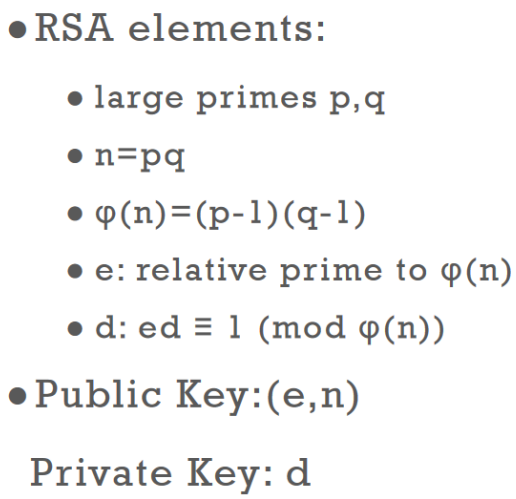

Fig. 7. RSA elements

- 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”).

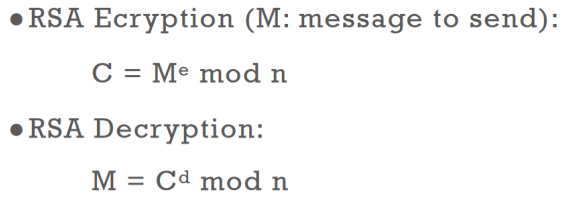

Fig. 8. RSA

E. Side Channel Attack

A side Channel attack (SCA) is also called a side-channel attack. The core idea is to obtain ciphertext information by encrypting various leakage information generated during the running of software or hardware.

In a narrow sense, side-channel attacks refer to non-invasive attacks targeting cryptography algorithms, which can be broken through the disclosure of side-channel information during the operation of encrypted electronic devices. Side-channel attacks in a narrow sense mainly include timing attacks, energy analysis attacks, electromagnetic analysis attacks targeting cryptography algorithms [1]. This new type of attack is far more effective than the mathematical methods of cryptanalysis and therefore poses a serious threat to cryptographic devices.

In a broad sense, intrusive, semi-intrusive, and non-intrusive attacks against security devices belong to the category of side-channel attacks. Side-channel attacks in the broad sense are often imaginative and have various attack methods, such as sound analysis attack, electromagnetic analysis attack, Wiley attack based on WiFi channel state. And attacks using state and process information through the kernel. [7]

IV. Technical Approach

To achieve our experimental goals, we decided to divide the whole process into four stages. They are: learn SGX first and try to use Teaclave. Secondly, learn the target two signature algorithms, and clearly understand their implementation details in the existing codebase. Third, migrate the two target signature algorithms from the code repository to TEE-SGX and try to compile them. Thirdly, we will learn the methods and details of channel detection attacks, and summarize them into a document, and try to find channel detection vulnerabilities through the assistance of two tools of choice. Finally, we will fix the problem audit the object code. and try to exploit the vulnerability to carry out the attack

A. Stage 1

According to our research methods, our first step is to apply Intel SGX and Teaclave. And we have finished these parts successfully.Since the installation and use of Teaclave are based on docker tools, we spent part of our time learning the establishment, use, deletion, and other functions of Docker. The skillful use of these tools provided us with considerable convenience in the subsequent experiments. 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.

B. Stage 2

According to our plan, our second step is to learn the mathematical principles and detailed implementation of the two signature algorithms.

1) Learning the mathematical process of algorithms: We spent about two week learning the specific mathematical process of the two signature algorithms and prepared a group meeting to share the knowledge we learned about cryptography with other students. The specific implementation process of the two algorithms has been introduced in detail in the background section above, so we will not repeat it here.

2) Learning the code implementation of algorithms: After that, we went back to an existing RUST encryption warehouse [4] to study its code implementation based on the mathematical process we learned. The main file structure of the repository is shown below, with the core code in the file "hazmat.rs".

In this code file, we can find the bottom two algorithms that sign and validate pre-hashed data. As you can see below, their implementation is almost identical to the mathematical process described above. But the code in this file is too bare and basic.

```rust
#[cfg(feature = "arithmetic")]
#[cfg_attr(docsrs, doc(cfg(feature = "arithmetic")))]
pub trait VerifyPrimitive<C>: AffineXCoordinate<C> + Copy + Sized
where
    C: PrimeCurve + AffineArithmetic<AffinePoint = Self> + ProjectiveArithmetic,
    Scalar<C>: Reduce<C::UInt>,
    SignatureSize<C>: ArrayLength<u8>,
{
    /// Verify the prehashed message against the provided signature
    ///
    /// Accepts the following arguments:
    ///
    /// - `z`: prehashed message to be verified
    /// - `sig`: signature to be verified against the key and message
    fn verify_prehashed(&self, z: Scalar<C>, sig: &Signature<C>) -> Result<()> {
        let (r, s) = sig.split_scalars();
        let s_inv = Option::<Scalar<C>>::from(s.invert()).ok_or_else(Error::new)?;
        let u1 = z * s_inv;
        let u2 = *r * s_inv;

        let x = ((C::ProjectivePoint::generator() * u1) + (C::ProjectivePoint::from(*self) * u2))
            .to_affine()
            .x();

        if Scalar::<C>::from_be_bytes_reduced(x) == *r {
            Ok(())
        } else {
            Err(Error::new())
        }
    }
}
```

Fig. 9. Verification method of ECDSA

```rust
    #[allow(non_snake_case)]
    fn try_sign_prehashed<K>(
        &self,
        k: K,
        z: Scalar<C>,
    ) -> Result<(Signature<C>, Option<RecoveryId>)>
    where
        K: Borrow<Self> + Invert<Output = Self>,
    {
        if k.borrow().is_zero().into() {
            return Err(Error::new());
        }

        // Compute scalar inversion of k
        let k_inv = Option::<Scalar<C>>::from(k.invert()).ok_or_else(Error::new)?;

        // Compute R = kG
        let R = (C::ProjectivePoint::generator() * k.borrow()).to_affine();

        // Lift x-coordinate of R (element of base field) into a serialized big
        // integer, then reduce it into an element of the scalar field
        let r = Self::from_be_bytes_reduced(R.x());

        // Compute `s` as a signature over `r` and `z`.
        let s = k_inv * (z + (r * self));

        if s.is_zero().into() {
            return Err(Error::new());
        }

        // TODO(tarcieri): support for computing recovery ID
        Ok((Signature::from_scalars(r, s)?, None))
    }
}
```

Fig. 10. Sign method of ECDSA

On top of this, the encryption warehouse carries on the higher level encapsulation to it, respectively

is ”sign.rs” and ”verify.rs”.

C. Stage 3

Our goal in the third step is to complete the code pruning and migration work, given that we have a detailed understanding of the overall structure of the code in the second step, the migration work is not difficult to do, can be said to be very smooth. Because of the compact structure of the object code repository and the fact that most of the code can be used in normal application scenarios, we had very limited pruning. After removing some of the test code, we carried out the migration. For our first contact with engineering problems related to transplantation, we referred to a brief document uploaded on Github shared by our classmates, which mainly introduced the steps needed for transplantation. [5]

D. Stage 4

Our last task was to learn how to detect channel attacks and defenses and to audit and modify the current code repository. We first read several papers and handouts related to side channel vulnerability detection.

We specifically learn a tutorial about the target algorithm ECDSA and RSA related side-channel detection, completed by Peter Schwabe on January 18, 2015. This section describes the common Timing Attach to resist RSA encryption. It is aimed at the widely existing operation of seeking power and then taking modulus in the RSA algorithm. Such operation is often reduced by Horner's Rule algorithm to reduce the complexity of calculation.

```
/* This really wants to be done with long integers */
uint32 modexp(uint32 a, uint32 mod, unsigned char exp[4]) {
  int i,j;
  uint32 r = 1;
  for(i=3;i>=0;i--) {
    for(j=7;j>=0;j--) {
      r = ((uint64)r*r) % mod;
      if(exp[i] & (1<<j))
        r = ((uint64)a*r) % mod;
    }
  }
  return r;
}
```

Fig. 11. Timing side channel vulnerability in RSA

After learning the above papers, we were still not clear about how to find channel vulnerability in complex code, so we prepared to borrow the existing channel vulnerability testing tools. With the help of the teacher, we found several different channel vulnerability detectors from past studies:

1) ctgrind: Ctgrind from Adam Langley from 12 years ago: The tool valgrind could check all the branches and memory accesses to make sure that they haven't been tainted with secret data. This would mean keeping track of every bit in memory to know if it's secret or not, likewise for all the CPU registers. Preferably at the bit level. The tool would also have to know that adding secret and non-secret data results in secret data etc. Ctgrind uses memCheck to treat our secret data as uninitialized. But this checker only supprot C/C++. [3]

2) SCALib: The SCAlib introduce themselves to be a python package that contains state-of-the-art tools for side-channel evaluation. It focuses on providing efficient implementations of analysis methods widely used by the side-channel community and maintaining a flexible and simple interface. But it mainly used for Python code. [6]

3) DATA: DATA which is Differential Address Trace Analysis is a dynamic side-channel analysis framework based on differential address traces. And the difference in DATA is that it analyzes at the binary level. And it's threat model is powerful since it assumes that the attacker can know the detail information of the binary for example, the instruction, the operands, and instruction address. However, there are some limitations of DATA which are the hardware bugs.

The DATA detection process consists of three steps. The first is difference detection phase, the second is leakage detection phase, and the third is leakage classification phase. The first phase is to record the execution of the tested binary program with different secret inputs. For each secret input, there is a corresponding trace which is the execution of the binary program. And then compare each trace to generate a difference report. The difference is about two types which are data accesses and branches. And this two type differences are both secret input dependent. The second phase is to improve accuracy and reduce the number of false negative and false positive. More detail, this part is

to filter the difference caused by the randomness of the binary program and the difference which are secret input independent. In order to achieve this goal, the binary program is executed with a fixed secret input and a set of random secret inputs. And, compare the address accesses between this two part, the fixed secret input and the set of random secret inputs. And there is a possible that the fixed secret input address accesses is similar to that of the set of random secret inputs. To avoid this, DATA repeats to choose multiple fixed secret inputs. And, the third part is to find the relation between the secret input and the leaks. In order to achieve this, researchers reconstructed the expression of secret input and information leak. For the secret input, there are two ways supported by DATA which are input slicing and Hamming weight Model [15]. The input slicing is suitable for symmetric ciphers and Hamming weight is suitable for asymmetric ciphers. But this is not absolute. After that, researcher used the Randomized Dependence Coefficient [14] to find the linear or non-linear relation between the secret input and the information leaks.

4) MIRAI: MIRAI is an abstract interpreter for the Rust compiler's mid-level intermediate representation (MIR). MIRAI have several use cases:

- MIRAI can be used as a linter that finds panics that may be unintentional or are not the best way to terminate a program.
- MIRAI can also be used to verify correctness properties. Such properties need to be encoded into annotations of the source program.
- MIRAI can be used to look for security bugs via taint analysis (information leaks, code injection bugs, etc.) and constant time analysis (information leaks via side channels).

We mainly use the third function to detect the vulnerability in the code. And this tool is easy to install and use. It also provide several test cases for us to learn how to use it. In order to test the channel, we need to add some tags into the code and write a test case. Then compile it with "cargo mirai –constant_time" and wait for the response.

V. Current progress and future plan

In the four weeks of work, we have made different progress in using the two tools to test the code. Next, we will summarize the weekly work progress, the current progress, and the problems we are facing. Finally, plan our work for the next few weeks.

A. DATA - Differential Address Trace Analysis

- 1) Compiled and Run the DATA tool. And checked the results of DATA run are consistent with those in the paper [20].
- 2) Tested RustCrypto Signature library with the secp256k1 (k256) elliptic curve. And A control leak is detected as show in 12. This method is used to do signature normalization, and this normalization keeps the size of the signature within a certain range. Thus, the control flow leak may leak some information about the secret key. After read the source code of the secp256k1, this leak information is viewed as a false positive. The reason is that this normalize step is executed to make the signature within a certain range after the sign process is done.
- 3) Tested RustCrypto crypto-bigint, this library is a big integer library that can be used in cryptographic applications. And this library is claimed that it is constant-time. A test about all the big integer operation supported by this library are tested. The debug version and the release version has different result. The reason for this is that the compiler optimization for debug mode and release mode are different. Debug mode contains some debugging information and release mode does not. In the end, we choose the release version to analysis. And the result shows that all the operations work well except for the wrapping_sqrt method which is to square root operation. Since there is some with the datagui program, I can just analyze the original XML file which is quite difficult. Because it is hard to see the source code that causes this information leak in XML file. For now, I can not identify if the information leak generated by DATA on the wrapping_sqrt method is false positive or not. After checking the source code, I found that the reason that

caused the wrapping_sqrt method information leak is that the wrapping_div method which is invoked. There is a problem that the wrapping_div method which pass test, however it occurs a information leak in the wrapping_sqrt method. And I found that in the source code comment, they claim that " Wrapped division is just normal division. There's no way wrapping could ever happen." However, I haven't figured out this relation.

- 4) Tested RustCrypto RSA, this library is a RSA implementation in pure rust. There are some challenges when using DATA analysis this library. First, the RSA secret input is not a single number but a group of numbers. This group contains decryption key, prime p, prime q and etc. To address this problem, I found the PEM file which can be used to send the private key. Another challenge is that analyze the RSA library with 2048 bits secret key need to too much memory and the performance and memory of my PC is limited. To address this challenge, the private key is limited to 512 bits. The reason is that the number of bits determines the amount of computation rather than the calculation logic. The main concern here is whether the implementation logic of RSA will cause the leakages of secret information. As for the result, there are many information leakages related to big number operation which are generated by DATA. To find out the reason, I check the RSA source code. As a result, I found that the RSA big number operation is not dependent on the crypto-bigin which is claimed constant-time.
- 5) Tested RustCrypto Signature library with the NIST P-256 (p256) elliptic curve. For this kind of implementation of ECDSA. There is no secret information leak for the release version.

B. MIRAI - Rust compiler's mid-level intermediate representation (MIR)

During the four weeks' work, I have learned the use of the tool MIRAI and its related source code in detail. However, due to the limited information and documentation of the MIRAI tool, I can only get a glimpse of its operating principle. The basic

```
pub fn normalize_s(&self) -> Option<Self> {
    let s = self.s();

    if s.is_high().into() {
        let neg_s = -s;
        let mut result = self.clone();
        result.bytes[C::UInt::BYTE_SIZE..].copy_from_slice(&neg_s.to_repr());
        Some(result)
    } else {
        None
    }
}
```

Fig. 12. Control flow leak in ecdsa algorithm with secp256k1

workflow is similar to that of a normal Static Analyzer, which traverses all possible paths through the entry point of a function and verifies in turn that no dead-end occurs. So when testing a library function, implement unit test code for a library function to provide an entry point for MIRAI. Secondly, it is based on the tag for taint analysis. That is to say, we need to tag the data that needs to be tracked in advance, and then set the case that the tag can propagate. For example, if value X has a tag that propagates via addition operations, then value X + 1 will also have that tag.

The MIRAI tool is based on the MIR level, which is also known as Rust's *Mid-level Intermediate Representation*. MIR is one layer in the compilation process between HIR and LLVM IR. This tool chose MIR to avoid some of the additional performance costs of recompilation.

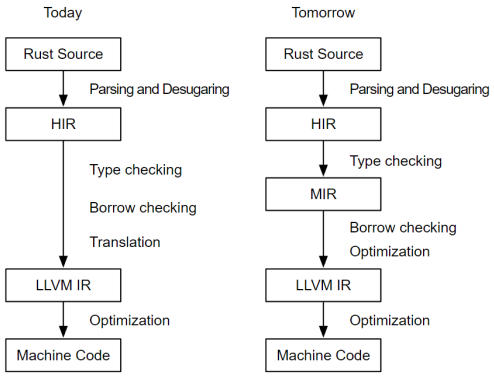

Fig. 13. MIR

In week 9, I completed a test code to call the ECDSA function in the RustCrypto library, which

passed the test in the Linux environment. After that, I put it into MIRAI for testing but found that MIRAI's support for Linux was not complete enough. After testing, the code could not compile normally. There is an error that occurs while exiting the test code. After discussing it with my teacher, I found that this problem was also mentioned in later documentation in MIRAI's repository. The reason for this is that z3, the other tool MIRAI relies on, doesn't work on Linux using its configuration. Tool official Z3 dependency in the way of configuration is given from "[Z3 distributions of prebuilt] (https://github.com/Z3Prover/z3/releases), and it can't come with MIRAI on Linux good work.

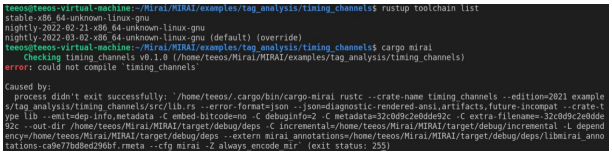

Fig. 14. error1

In week 10, I tried to solve the problem I encountered in Week 9. Since z3 cannot run normally in normal configuration mode, the author suggests that users should try to manually clone the z3 repository source code, and manually build the installation. This was similar to the discussion with the teacher, so I completed the manual build of Z3 and started a new test according to this step. After the test, I found that the last error had been solved normally, but the test still could not proceed, the error message was pointed out (" Process didn't exit successfully "), and the error location could not be displayed normally. Later, when browsing related solutions in the issue under the repository, I found that other Linux users had similar problems. In the related issue, the user also encountered this problem when conducting validation tests in the CI of the Github repository. The author also pointed out that he could pass the tests on macOS. Finally, the issue did not provide a good solution for Linux. The user also just by changing the system to solve the problem. Since I didn't have a macOS computer, I asked the teacher to help me do a test. Although he still had bugs in the program on macOS, the bug report could be displayed normally. So I decided to install a macOS virtual machine environment locally and test it out.

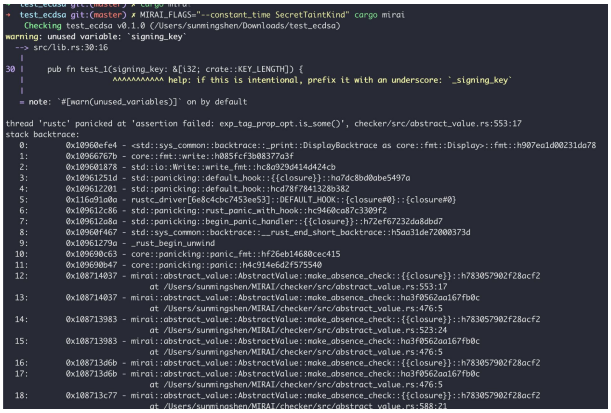

Fig. 15. error2

In week 11, I installed a macOS virtual machine environment locally, but because VMware itself does not support macOS virtual machines, I had to install an older macOS environment in some way. I also set up the MIRAI runtime environment in that VM and continued testing. The test still failed, but an error message indicated that the problem was some code in the Checker package in MIRAI's source code. I found the error source in this code and tried to understand its meaning. Since there was no annotation in the code and the author did not provide documentation about this package, I could only guess that it was an error generated during tag Propagation of tagged data. In order to verify that the problem was not caused by the environment, I changed the test code and requested the teacher to conduct the test under the macOS of the real computer. The test results indicated that the problem was caused by insufficient memory. I was very surprised that there was no memory shortage on a real machine with 16GB of memory, but not on a virtual environment with 4GB of memory. It was difficult to locate the problem. Later, when looking through relevant materials in his issue, I found that the author also had the problem of insufficient memory in the real computer environment of 32GB, and he also pointed out that this tool does consume a lot of memory. It has been difficult to advance the testing work due to the difficulty in solving the problem of hardware performance. Therefore, I have been discussing with my teacher to find a new solution or choose a new tool.

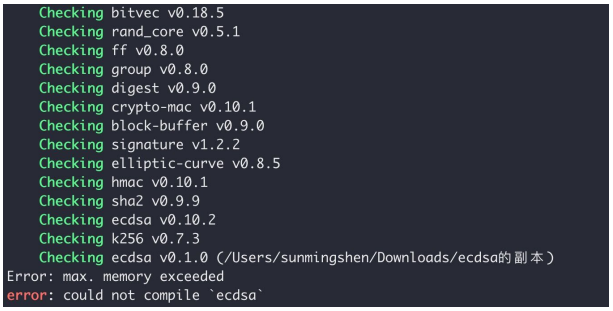

Fig. 16. error3

C. SGX - Side Channel

To figure out whether the information leakage generated by DATA in normal case is also information leakage in SGX mode, we did some research on the side channel in SGX mode. We want to know whether SGX has additional effect on the side channel, what is the effect, is it enhanced side channel, or weakened side channel. And we also want to know is there some technique that obtain the information that needed in DATA threat model in SGX mode.

The result we found is that the SGX does not provide additional protection for side channels and on the Intel SGX's website, they said that "Preventing side channel attacks is a matter for the enclave developer." [1]

The side channel attack technique in SGX mode are as follows. [19]

Cache attack There are two kinds of method Flush+Reload and Prime+Probe. The Flush+Reload can not work well since the enclave does not share memory with other process. However, the Prime+Probe can be used. [12]

TLB Although Flush+Reload can not work on enclave in Cache attack, but van Bulck used Flush+Reload to attack TLB managed by OS. [10]

Controlled-Channel Attack The Controlled-Channel attack is a new attack on Intel SGX proposed by Yuanzhong Xu. [21] This attack suppose that the Operating System is not trusted and the OS controls the mapping of virtual pages and physical pages for enclave. Thus, the OS can set the present bit in the page table entries. In the way, the page fault can be triggered. Therefore, when the enclave accesses unmapped pages, the page fault will be triggered and the OS will handle this page fault. By doing this, the OS can get the information of enclave memory accesses at page level.

Interrupt-driven Attack Marcus Hähnel [13] implemented the Interrupt-driven attack by taking advantage of frequent timer interrupt of the Advanced Programmable Interrupt Controller (APIC) to pause the execution of enclave with the asynchronous exit event (AEX). What's more, by modifying the access bit of enclave's PTE, they can get the information of single-step page-table access when enclave is executing. And they attacked a string compare function. In the end, they got the information leak at byte granularity. More important, the interrupt-driven attack is improved to single instruction level with zero noise in practise by [18] and [16]

As a result, not all the attacks above can work well in SGX mode. However, the Interrupt-driven attacks work well and it is instruction level which means we can get all the information during execution of enclave which satisfy the threat mode of DATA. That means, the information leakages generated by DATA is meaningful in SGX mode.

There is a problem which is that it takes a lot of memory to test encryption algorithm with MIRAI and we don't have enough hardware resources. Thus, we can not compare the result between MIRAI and DATA. To address this problem, we took a tricky method. The method is that we first analyze the report generated by DATA and select some fatal leakages which are usually small piece of code. Second, test these fatal leakages with MIRAI and find out whether MIRAI can generate the same result as DATA.

The answer to the above question is YES. We test a few code fragments which are shown to have control flow side-channel vulnerability on DATA.

The first one is a big integer multiply function used by RSA, and the picture only shows small segments of the whole code. This fragment is extracted from the original code and could represent the problem. This function is generally about doing a bit-wise multiplication operation. It first matches the format of input and then does different operations

toward it.

```
pub enum Sign {
    Minus,
    NoSign,
    Plus,
}

pub mod non_constant_time {

    pub fn mul(this: crate::Sign, other: crate::Sign) -> crate::Sign {
        precondition!(has_tag!(&this, crate::SecretTaint));

        match (this, other) {
            (crate::Sign::NoSign, _) | (_, crate::Sign::NoSign) => crate::Sign::NoSign,
            (crate::Sign::Plus, crate::Sign::Plus) | (crate::Sign::Minus, crate::Sign::Minus) => crate::Sign::Plus,
            (crate::Sign::Plus, crate::Sign::Minus) | (crate::Sign::Minus, crate::Sign::Plus) => crate::Sign::Minus,
        }
    }
}
```

Fig. 17. multiply of sign

The test result is in the below picture. It shows that MIRAI could also find out that this fragment has a side-channel vulnerability.

```
teeos@teeos-virtual-machine:~/MIRAI_0411/test_0602/test1$ cargo mirai
    Checking mirai-annotations v1.12.0 (/home/teeos/MIRAI_0411/MIRAI/annotations)
    Checking test1 v0.1.0 (/home/teeos/MIRAI_0411/test_0602/test1)
warning: the branch condition may have a SecretTaintKind tag
  --> src/lib.rs:38:9
   |
38 |         match (this, other) {
   |         ^^^^^^^^^^^^^^^^^^^

warning: `test1` (lib) generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 2.83s
```

Fig. 18. multiply of sign in MIRAI

The second one is a function of adding two big integers which are represented in a self-defined type ”BigDigit”. This function is also extracted from the original code. This function was originally designed to calculate the addition of two numbers a and b along with the previous carry bit.

```
#[inline]
pub fn adc(a: BigDigit, b: BigDigit, acc: &mut DoubleBigDigit) -> BigDigit {
    *acc += a as DoubleBigDigit;
    *acc += b as DoubleBigDigit;
    let lo : u32 = *acc as BigDigit;
    *acc >>= BITS;
    lo
}

pub fn __add2(a: &mut [BigDigit], b: &[BigDigit], carry: &mut u64){
    precondition!(has_tag!(a, crate::SecretTaint));
    precondition!(has_tag!(b, crate::SecretTaint));
    precondition!(has_tag!(carry, crate::SecretTaint));

    debug_assert!(a.len() >= b.len());

    // let mut carry = 0;
    let (a_lo : &mut [u32], a_hi : &mut [u32]) = a.split_at_mut( mid: b.len());

    // for (a, b) in a_lo.iter_mut().zip(b) {
    //     *a = adc(*a, *b, &mut carry);
    // }

    if *carry != 0 {
        for a : &mut u32 in a_hi {
            *a = adc(*a, b: 0, acc: carry);
            if *carry == 0 {
                break;
            }
        }
    }
}
```

Fig. 19. addition of two big integer

The test result shows that the side-channel vulnerability happened when it judges whether a carry bit is zero or not.

```
teeos@teeos-virtual-machine:~/MIRAI_0411/test_0602/test2$ cargo mirai
    Checking test2 v0.1.0 (/home/teeos/MIRAI_0411/test_0602/test2)
warning: unused variable: `a_lo`
  --> src/lib.rs:79:14
   |
79 |         let (a_lo, a_hi) = a.split_at_mut(b.len());
   |              ^^^^ help: if this is intentional, prefix it with an underscore: `_a_lo`
   |
   = note: `#[warn(unused_variables)]` on by default

warning: the branch condition may have a SecretTaintKind tag
  --> src/lib.rs:76:9
   |
76 |         debug_assert!(a.len() >= b.len());
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warning: the branch condition has a SecretTaintKind tag
  --> src/lib.rs:85:12
   |
85 |         if *carry != 0 {
   |            ^^^^^^^^^^

warning: `test2` (lib) generated 3 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 2.20s
```

Fig. 20. addition of two big integer in MIRAI

VI. Time Line

- Currently: We have installed and learn to use two tools: ”DATA” and ”MIRAI”. Also, we have already test some algorithms with DATA.
- Until Week 9: Try to find vulnerability with both two tools and figure out how to trigger the bugs. Then distinguish whether they are real bugs.
- Until Week 11: Fix the bugs and test again.
- After Week 11: If the progress is optimistic, we will try to exploit the found vulnerability.

References

[1] Intel® sgx and side-channels.

[2] blog of tool teaclave, 2021.

[3] ctgrind, 2021.

[4] Rust crypto library, 2021.

[5] Rust sgx transplant process, 2021.

[6] Scalib, 2021.

[7] Side channel attack - baidubaike, 2021.

[8] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols.

[9] Benger, van de Pol, Smart, and Yarom. “ooh aah... just a little bit”: A small amount of side channel can go a long way.

[10] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page Table-Based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, August 2017. USENIX Association.

[11] Victor Costan and Srinivas Devadas. Intel sgx explained.

[12] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 299–312, Santa Clara, CA, July 2017. USENIX Association.

[14] David Lopez-Paz, Philipp Hennig, and Bernhard Schölkopf. The randomized dependence coefficient. *Advances in Neural Information Processing Systems*, 04 2013.

[15] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. 01 2007.

[16] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled Instruction-Level attacks on enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 469–486. USENIX Association, August 2020.

[17] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2009.

[18] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 178–195, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Samuel Weiser. *Enclave Security and Address-based Side Channels*. PhD thesis, June 2020.

[20] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. Data - differential address trace analysis. In *Krypto-Tag*, 2018.

[21] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[22] Yarom and Falkner. Flush + reload: a high resolution, low noise, l3 cache side-channel attack.