Romeo: SGX-Based Software Anti-piracy Framework

Yanning Du(✉), Xin Song, and Yichuan Wang

Xi'an University of Technology, Xi'an, Shaanxi, China
duyanning@gmail.com

Abstract. Preventing software piracy has always been a concern of software developers. Since crackers can track and analyze the application code, any client-side anti-piracy mechanism can only increase the cost of crackers, but cannot really stop them, unless the anti-piracy mechanism is put on the server along with the core functionality of the software. However, this approach harms the user experience. In this paper, we propose a software anti-piracy framework that makes it possible for developer to integrate anti-piracy mechanisms into the client-side without compromising the user experience through the use of Intel's SGX technology.

Keywords: Anti-piracy · SGX · Enclave

1 Introduction

As an application developer, it is always a headache to protect your rights against software piracy. You cannot protect your rights just by a licensing agreement. You must also use technical means to defend your rights from infringement.

It is common practice to include anti-piracy logic in the application code, which detects piracy and refuses to work when piracy is detected. However, both the logic for detecting piracy and the logic for denying service to pirated copies run the risk of being bypassed by crackers [1, 2, 4–9].

The source of this risk is that, in order to provide services to users, our software, i.e., the code that provides services to users, is distributed to users, along with the code that detects piracy and refuses to provide services.

This means that crackers have access to these codes. You cannot hide any anti-piracy logic in the public code. Even the most sophisticated anti-piracy logic, once presented to the cracker, is breached.

Distributing software to a user means presenting anti-piracy logic to crackers. In order to prevent crackers from tracking the anti-piracy logic, many software developers are forced to adopt a "kill a thousand enemies, lose eight hundred" approach. In this approach, developers place only part of the services in the app distributed to users, and place the rest of the services, along with the anti-piracy logic, on a server that crackers

Q. Jiang et al. (Eds.): SPNCE 2022, LNICST 496, pp. 140–155, 2023.
https://doi.org/10.1007/978-3-031-30623-5_10

cannot touch. Since crackers cannot modify the code on the server, they cannot breach the anti-piracy logic.

However, this approach is a way to sacrifice the user experience in exchange for the benefit of software developers. Because this approach not only puts the anti-piracy logic on the server, it also requires that the code that provides services to users must also be placed on the server. The premise that the logic for denying service to pirate users can work is that the right to decide whether to provide service to users is in the hands of the software developer. When the software developer decides to refuse service, the user does not get the service.

The services that are placed on the server must be critical, because if they are not, the cracker may choose to throw them away. However, putting critical functionality on the server can seriously impact the user experience.

Intel's Software Guard eXtensions (SGX) technology gives app developers another possibility. With this technology, developers can achieve the same anti-piracy purpose without putting the anti-piracy logic on the developer's server. The anti-piracy logic is included in the app together with other functions of the software, distributed to the user and installed on the user's computer. Thanks to the protection of SGX technology, although the anti-piracy logic is installed on the user's machine, it is not visible to crackers. Thus, the purpose of protecting the anti-piracy logic is achieved.

In short, we have made the following contributions in this work: We have demonstrated the application of SGX technology in anti-piracy. We have designed an anti-piracy scheme. We have designed a prototype framework for integrating SGX technology into application software.

2 Background

2.1 Anti-piracy Mechanism

The anti-piracy logic of software consists of two parts: the first part is the piracy detection logic, and the second part is the denial-of-service logic. An attack on either part by a cracker will cause the software's anti-piracy mechanism to fail.

The piracy detection logic is the code that determines if a copy is pirated. If this code is deployed on the user's computer, a cracker can analyze it in a cracking environment (e.g., debugger, IDA) to figure out how the piracy detection logic works, and then bypass the piracy detection logic or cheat the piracy detection logic into believing the pirated software is genuine software.

Denial of service logic is the ability to deny service when the piracy detection logic determines that the copy is a pirated copy. Similarly, a cracker can analyze this part of the code and, after figuring out its logic, modify it. So that it no longer works in the way the application developer intended.

Some developers chose to scatter piracy detection logic and denial-of-service logic throughout the application to deal with crackers. However, this approach is essentially the opposite of the modularity advocated by software engineering. Once you implement the piracy detection logic and denial of service logic as a function in accordance with the idea of software engineering, and then call them from time to time in the software's

business processes, these functions themselves will become the target of attack. If it is not implemented as a function, but directly scattered in the form of inlined code, it is also easy to find out by crackers using automated tools due to the same code pattern. It is very difficult for software developers to repeat the anti-piracy logic in different forms in every needed location, and it also interferes with the development of the software.

2.2 Software Guard Extensions

SGX technology [3–18] is an extension of the Intel processor. It provides a set of instructions through which an application can place a portion of its code in a memory area called an enclave. The enclave is still part of the process address space, but the content stored in the enclave, whether code or data, is not visible outside the enclave. This "outside" includes not only other parts of the process to which the enclave belongs, but also privileged software such as the operating system.

The data and code to be put in the enclave are encrypted and cannot be read by anyone until they are decrypted in the enclave. Only Intel processors that support SGX technology can decrypt them. The decrypted code and data are placed in the enclave where no one can access them. That is, nothing can access them except the code itself in the enclave.

Developers do not need to use the SGX instructions provided by the processor directly. They just need to call the SDK functions provided by Intel to utilize SGX [19–23]. The code and data to be put in the enclave is made into a dynamic library and distributed to the user in an encrypted form. The code and data in this dynamic library are decrypted by the processor and placed in the enclave. We can simply think of this dynamic library as existing in the enclave. The boundary between the enclave and the outside world is controlled by SGX. For the world outside the enclave, whether it is another part of the enclave's process or privileged software like an operating system, the code and data in the enclave seem like to exist in the memory of a remote computer, and cannot be touched directly. You can only call functions in the enclave indirectly by calling proxy functions, just like calling remote procedures. Here, as with RPCs, there is also a marshalling and unmarshalling of parameters. But by using the SDK from Intel, we can simply describe these functions in the Enclave Description Language (EDL), and the tools in the SDK will automatically generate the corresponding marshalling/unmarshalling code for us. The code in the enclave can also call external code, such as API functions provided by the operating system for network communication.

3 Overview

The design of anti-piracy mechanism is related to the following two aspects:

- The designer's vision of the user experience.
- The designer's vision of the developer experience.

The following is our view of these aspects. Finally, the basic working process of the framework and remote attestation is described.

3.1 User Experience

The following is an example to illustrate the user experience when using the application protected by Romeo framework.

The user has purchased the application and the license allows 3 instances to run simultaneously. The user has installed the application on four computers. When the user runs the application on the first, second and third computer in sequence, everything works fine. Keep the instances on these computers running, and then start the application on the fourth computer. At this point, the total number of instances running exceeds the maximum allowed by the license. However, the program starts normally on the fourth computer, without any complaint. But then the application on the first computer starts to strike. A dialog box pops up, telling the user that the number of instances running at the same time exceeds the limit allowed by the license the user purchased. When the user finds the dialog on the first computer, he restarts the application, and after the restart, the application works again, but then the application on the second computer goes into a strike.

That is, when the number of simultaneously online instances reaches the maximum allowed by the license, starting a new instance will cause the oldest instance to go into a strike state. The new instance does not tell the user that the number of simultaneously online instances has exceeded the limit. This solution penalizes a genuine user who has copied his software to others, thus creating the fact of piracy. For the genuine user, a new instance started by a pirate user will cause the genuine user's instance to strike. The user only knows that their rights have been compromised by the presence of pirate users. But he doesn't know which pirate user caused the problem.

3.2 Developer Experience

For the application developer, the software he sells to each user is the same. The only difference is the enclave DLL. In fact, the enclave DLL is basically the same from the point of view of source code, only the user ID (such as the user's email address provided when registering) string is different.

The process of regenerating the enclave DLL once for each user ID can be automated as part of the user registration process. When the user has completed registration and paid, the user downloads the resulting enclave DLL as an electronic license and copies it to the application directory.

This process is very easy for the developer to accomplish, whether he chooses to provide a registration system or to register the users manually himself. Because each time you only need to replace the user ID part in the source code of the enclave DLL, and then recompile to generate the enclave DLL.

The application developer needs to run a server. Each instance of the application communicates with this server so that the server can know how many instances of each sold copy are currently running. When the number of instances running simultaneously exceeds the maximum allowed by the license, the server notifies the oldest instance to go on strike.

3.3 Basic Working Process

When a critical service function of an application is executed in an enclave, the application sends a confession message to the server. The *confession message* consists of a *user ID*, a *random number*, and a *secret code*. The secret code is assigned to the instance by the server when the instance first confesses to the server. The confession message is sent in plaintext via a UDP packet.

When the server receives the message, it extracts the user ID and random numbers from the message and uses them to construct the *reassurance message*. The server encrypts the reassure message with its own private key $\{n, d\}$ and sends it back to the application instance.

$$E = R^d \bmod n \tag{1}$$

where E is the *encrypted reassurance message* and R is the *reassurance message*. R is the concatenation of the *user ID*, a *random number*, and an optional *secret code*.

$$R = id \| nonce \| secret^{opt} \tag{2}$$

Because the *reassurance message* is small, we encrypt it directly with the server's RSA private key. The customary hash process of signing is omitted.

After encryption and before it is transmitted over the network, the encrypted reassurance message is transformed from a binary block to a string to suit the requirements of JSON.

$$E' = mapping \cdot grouping \cdot serializing\ E \tag{3}$$

The function *serializing* is used to obtain the binary data block of E. The function *grouping* divides the binary data block into a grouping of 6 bits, and the function *mapping* maps each group to a printable character. The actual transmission is E'. Accordingly, an inverse transformation is performed when the reassurance message of this form is received.

$$E = unserializing \cdot merge \cdot unmapping E' \tag{4}$$

where *unmapping* converts E' from printable characters to 6-bit tuples, function *merge* combines these tuples, and *unserializing* restructures the merged binary data block.

Upon receiving the *reassurance message*, the instance decrypts it in the enclave with the server's public key $\{n, e\}$.

$$D = E^e \bmod n = \left(R^d\right)^e \bmod n \tag{5}$$

where D is the *decrypted reassurance message*, E is the *encrypted reassurance message*, and R is the *reassurance message*.

Accordingly, the verification process is simplified. We only need the RSA public key of the server to decrypt the received encrypted reassurance message.

The user ID and random number are extracted, and then compared with the user ID and random number of the previously sent *confession message*.

If the message contains the same information as in the previously sent confession message, the instance is validly reassured. Otherwise, the reassurance is considered invalid.

Effective reassurance causes a counter inside the enclave (which we call the *disappointment counter*) to be cleared to zero. Each execution of a critical service function of the application causes the disappointment counter to be incremented by one. When the value of the disappointment counter grows to a threshold due to the lack of timely reassurance messages, the critical service functions in the enclave will go on strike. This threshold is called the *heartbreak value.*

3.4 Remote Attestation

SGX provides integrity of code and confidentiality and integrity of data at run-time. However, it does NOT provide confidentiality of code offline as a binary file on disk. Adversaries can reverse engineer the binary enclave DLL. An adversary could disassemble it and then make a copy that bypasses checking for reassurance messages.

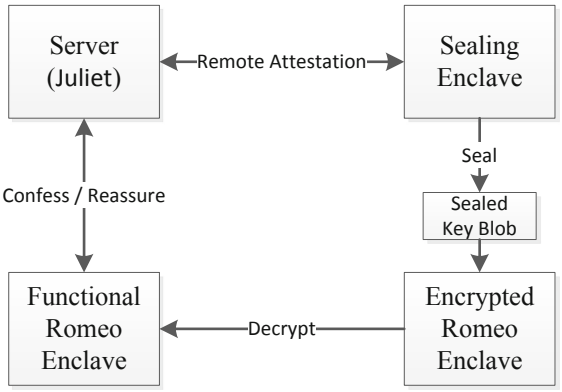

Fig. 1. Remote attestation

To solve this problem, we devised a sealing enclave in addition to the original enclave and called the original enclave Romeo Enclave (see Fig. 1). The Romeo Enclave DLL is provided to the user in an encrypted form that needs to be decrypted before it is run for the first time, with the key coming from the server. The sealing enclave communicates with the server and uses SGX's remote attestation mechanism to obtain the decryption key from the server, seals the key and saves it to a file. The client app reads this sealed key from the file and unseals it, then uses this key to decrypt itself (via SGX's Protected Code Loader) and restore itself to a functional Romeo enclave. The remote attestation is performed only once in the entire process. After that, Romeo Enclave communicates directly with servers, eliminating the need for remote attestation.

4 Implementation

The Romeo framework can be divided into two parts: the client part (which can be called Romeo), and the server part (which can be called Juliet).

4.1 Client

The client part is a library. The application code needs to be modified slightly to use this library. However, these modifications are quite easy. There is no impact on the logical structure of the application.

First, the developer needs to identify some core functions in the application and place them in the enclave for execution. These core functions should meet the following conditions:

- These functions are located on the critical path of the critical services provided by the application. We will achieve denial of service by controlling these critical functions.
- They are frequently invoked. Only then can we maintain the power to threaten pirate users with it on a regular basis. If you cannot locate such a function, it's fine to locate a collection of functions, as long as the functions in that collection are called frequently.
- The behavior of these functions is more complex. The relationship between inputs and outputs is difficult to determine. If it is too simple, the cracker will analyze the relationship and replace it with a function written by the cracker itself, which never strike.

Then, we modify these key functions, as in Fig. 2:

- Insert the code to make a confession to the server (Juliet). The first message sent contain only the user ID and a random number. Subsequent messages will contain, in addition to the user ID and random number, a secret code. This secret code is assigned by the server. After the server receives the first confession message from the client, it sends a reassurance message to the client. The reassurance message contains a secret code. The client will include this secret code in all future messages. The client will not send a new confession until the last one has been reassured by a confession message.
- Insert the strike logic. Each execution of the strike logic will add one to the *disappointment counter*, and when the value of the disappointment counter exceeds the heartbreak value, the client will enter the heartbroken state. Once it enters the heartbreak state, the enclave will refuse to execute core functions. After that, even if you receive a reassurance message from the server later, it does not help. In other words, once Remote's heart is broken, there is no way to recover.

In addition to these modifications of core functions in the enclave, the following additions need to be made to the code outside the enclave:

- Listen for reassurance messages from the server. Upon receiving the reassurance message, transport it into the enclave.

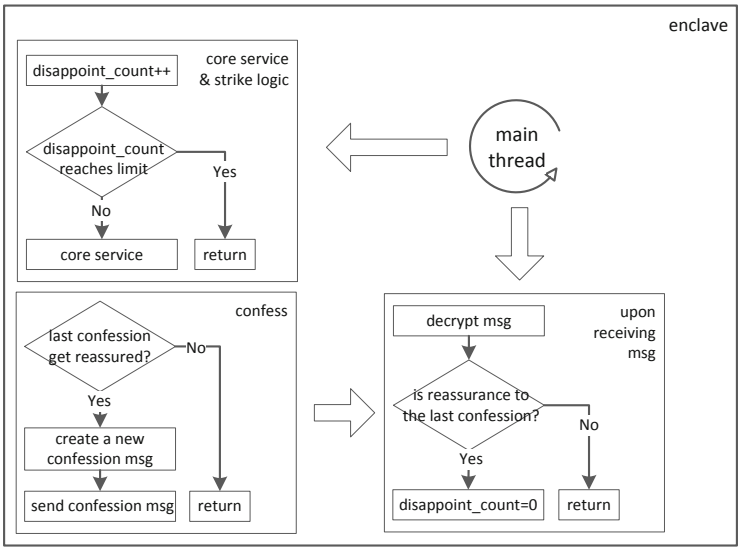

Fig. 2. Structure of client.

Then add code to the enclave to do the following:

- The enclave gets the incoming reassurance message from the outside world and decrypts the reassurance message with the server's public key.
- Extract the user ID and random number from the reassurance message, and then compare them with the user ID and random number from the previously sent confession message.
- If the information contained in the reassurance message is the same as the message sent in the last confession message, then the instance is validly reassured. Otherwise, the reassurance is considered invalid.
- Effective reassurance will cause the disappointment counter inside the enclave to be cleared.

4.2 Server

The server part is a separate program. We call it Juliet. Juliet's role is to receive confession messages from the client, and to send the appropriate reassurance messages. By controlling whether or not to send a reassurance message to Romeo, we can control whether or not to put Romeo into a heartbroken state. This allows us to control whether or not the application refuses to provide service (i.e., strike).

The confession message sent by the client contains the user ID. The user ID uniquely identifies each copy of the software.

As in Fig. 3, in the server, there is a list of users. The user IDs that appear in this list are the legitimate user IDs. Some IDs are in an expired state, which means that they purchased the software as a service, i.e. they can only use the software for a limited period of time. After the service period has expired, the software can be retained, but

the software will refuse to work because it does not receive a reassurance message from the server.

For each user ID in the user list, there is a list of instances associated with it. The instance list holds the IDs of all online instances of the application sold to the user ID.

The element in the instance list contains two fields: instance ID and secret code.

The instance ID is the < *IP address:port* > of the client. It can be derived from the UDP packet.

1. Upon receiving the confession message from the instance, the server extracts the user ID of the sender from the confession message and checks whether the user ID is in the *user list*:
2. If it is not in the user list, it is an illegal user and no reassurance message will be sent.
3. If it is in the list, check if the service period is exceeded, and if it is, no reassurance message will be sent.
4. The server extracts the *IP address* and *port number* from the UDP packet which carries the confession message, and combines the two as the *instance ID*. And then look up the instance ID in the *instance list*.

If it does not exist, this may be due to one of the following conditions:

– It is a new instance.
– It may be a confession from old instances that have been neglected. These old instances have already been taken offline because of the start of new instances. But they don't know it yet, and send a confession message. The message will be neglected.
– This is an old instance that restarted after a strike. This situation is actually no different from a new instance starting up.
– At this time the server only needs to see whether the confession contains a secret code to do different processing:
– If the confession does not contain a *secret code*, then the confession is from a new instance (or an old instance after a restart, which is essentially a new instance). All it has to do at this point is to generate a new instance structure (which contains the newly generated secret code), add the instance structure to the instance list, and send a reassurance message, which contains the newly generated secret code, to the instance.
– If the message contains a secret code. This means that the confession message is sent by an old instance that had been neglected. Just ignore it.

4. If the instance ID exists in the instance list, it checks if the confession contains a secret code.
5. If not, this is a very strange phenomenon. While it's not the first time you confess, why would there be no secret code, the server will think this situation as a fake message sent by adversaries. Just ignore it.
6. If so, then compare the secret code in the confession with the secret code saved in the instance structure in the instance list.

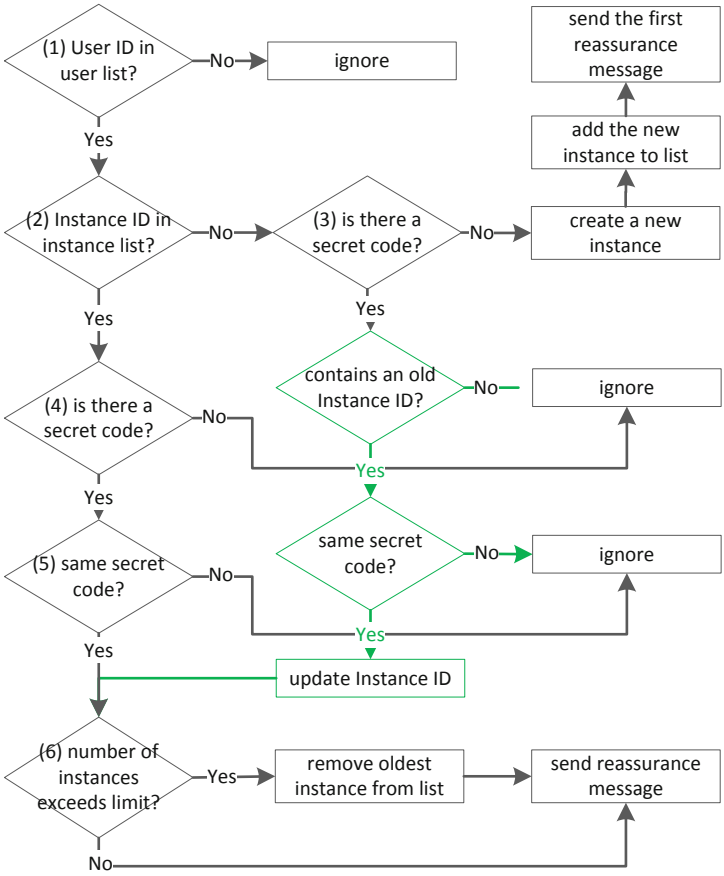

Fig. 3. Server-side workflow.

7. If the code doesn't match, which means you don't even know the secret code, you must be the adversary. Just ignore it.
8. If the secret code is the same, it means that it is a legitimate user and a legitimate instance. At this point, the server will look at the length of the instance list to determine whether the number of active instances exceeds the upper limit of the license.
9. If the upper limit is exceeded, the oldest instances are removed from the instance list. Then the server sends a reassurance message to the client. The reassurance message contains the user ID, a random number from the confession message, and the secret code. This secret code does not change. The reassurance message is encrypted using the server's private key before it is sent to the client as a UDP packet.

5 Experiment and Analysis

To demonstrate the advantages of adopting the Romeo framework, we developed a mind mapping program and then modified it to include anti-piracy logic.

The mind map program is written in C++ to facilitate the use of the Intel SGX SDK, which is provided as a C/C++ library. In order to provide a graphical user interface, the program uses the Qt framework. The server-side code is written in Java. The data between the client and server is passed in the form of JSON to ease the difficulty of communicating across languages.

All code (including *Juliet* the server, *Romeo* the library, as well as the demo application *diagram*) can be found at https://github.com/duyanning/romeo.

5.1 Experimental Environment

Table 1 shows the specific information of the development tools and platform used in the experiment. The most important of these is the version of SGX, as SGX varies significantly from version to version and some functions and libraries may be renamed or removed.

Table 1. Experimental configuration.

Location	Software	Version
Server side	Operating system	Ubuntu 16.04.7 LTS
	Java	java version “11.0.15” 2022-04-19 LTS
Client side	Operating system	Windows 10 21H2 19044.2251
	C++	Visual Studio Community 2019 16.11.18
	Qt framework	Qt 5.15.2 for MSVC 2019 (64-bit)
	Intel SGX	Intel SGX SDK for Windows 2.15.100.4

5.2 Performance Evaluation

The comparison experiment was conducted between the two versions. In one version, a traditional server-centric anti-piracy solution is adopted, in which user-generated mind map files are saved on the server. Although users can export them to local machine, they must be re-imported to the server for viewing and editing. The other version uses our Romeo framework, and the mind map file is saved locally and can be viewed and edited locally.

To compare the performance of the two solutions, we measured the time taken and the amount of data transferred for the five most common mind map operations (*add branch*, *remove subtree*, *rename branch*, *move subtree*, *relayout*).

In the traditional server-centered solution, any operation on the mind map will trigger the network communication between the client and the server, and the data transmitted is

closely related to the business logic of the mind map software itself, and the data amount of this kind of information is large. In a scenario using the Romeo framework, the data transferred has nothing to do with what the client software does. The data transferred is the *confession message* and *reassurance message* used for anti-piracy monitoring, and the amount of data is small. And the manipulation of the mind map only triggers network communication with the server with a certain probability. The client does not send a confession message to the server again when the previous message has not received the corresponding reassurance message.

The *confession messages* are sent in clear text while the *reassurance messages* are sent in encrypted form. The content of the confession message is generated in the enclave, but is delivered by untrusted code located outside the enclave. Decryption and verification of the comfort message must be done in the enclave. Because the amount of data decrypted at a time is very small, and the confession messages are sent as non-blocking UDP datagrams, the Romeo framework's interference with the smooth operation experience of the user of an interactive application is minimal and almost imperceptible.

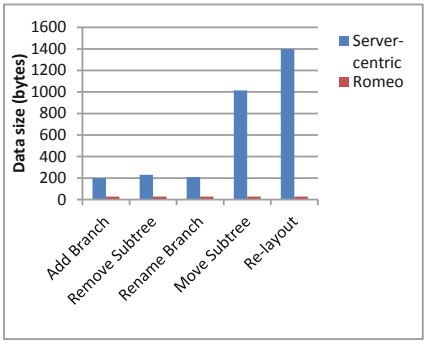

Fig. 4. The amount of data that needs to be transferred to perform a critical operation.

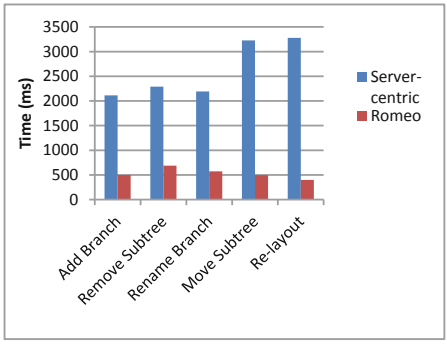

Fig. 5. Time spent to perform a critical operation.

Figure 4 shows the amount of data passed between client and server when various key operations are performed. As you can see, in the version that uses the Romeo framework, the amount of data transferred between the client and server is a constant (which is equal to the size of the confession message plus the reassurance message). In the traditional server-centric version, the amount of data transferred depends on how much different operations change the mind map. Of these, the re-layout operation results in the most staggering amount of data transfer. Also, for layout operations, the amount of data transferred is proportional to the size of the mind map.

Figure 5 shows the time it takes to perform various operations. As you can see, the version with the Romeo framework takes less time. This time is only related to the amount of data changed and the read/write speed of the local disk. In the traditional server-centric version, the time is related to the time spent reading and writing from the server's hard disk, plus the latency associated with network transmission. Also, when

the server is serving multiple clients, this time overhead is even greater. And it's not a constant. The busier the server, the longer it takes.

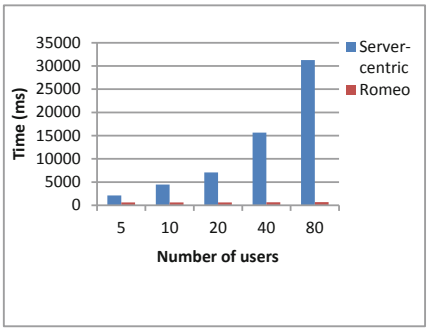

Fig. 6. Scalability as the number of users increases.

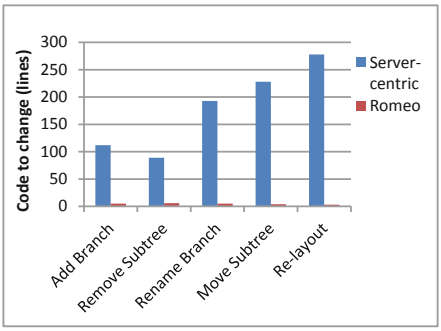

Fig. 7. The amount of lines of changes made to the code

Figure 6 shows the time taken to re-layout the mind map as the number of concurrent online users increases. We can see that in a traditional server-centric scenario, the time spent increases linearly with the number of users. This means that more hardware resources must be provided as the number of users increases. In the case of the Romeo framework, this time is almost a constant. This is because the processing of the confession message and the sending of the reassurance message cost very little.

Figure 7 shows the number of lines of code that need to be modified under different approaches. As you can see from this figure, the traditional server-centric version requires a deep refactoring of the code. This refactoring not only changes the logic of the client code, but also adds a lot of application-specific logic to the server, essentially splitting the application into two parts, one running on the client and the other on the server. The Romeo framework, on the other hand, requires very few changes to the code, and these changes are trivial, that is, they do not require substantial changes to the applied logic. And the same server can support multiple software protection.

Table 2 lists potential attacks and how the Romeo framework deals with them. As this table shows, the protection provided by the Romeo framework cannot be breached by every possible attack type we can think of.

Table 2. Potential attacks and corresponding solutions.

Attack	Solution
The adversary creates a cracking environment to breach the strike logic	By placing the strike logic in the enclave along with the core functionality, the adversary cannot bypass the strike logic
The adversary uses a fake server to send a fake reassurance message	Since the fake server does not know the private key of the real server, i.e., Juliet, it cannot mock an reassurance message that can reassure Romeo
By monitoring the network traffic of an instance for a certain period of time, the adversary learns what kind of confession message requires what kind of reassurance message	It is difficult to determine the relationship between the confession message and the reassurance message because the confession message contains a random number, and the random number has a wide range of variation
The confession message sent by the cracked client contains the ID of another user, and this ID will keep changing, but it's always a legitimate ID, so as to rub other users	The user ID of a reassurance message that the cracked client get by sending a confession message containing the ID of another user is different from the user ID that the enclave expects, and the enclave will compare this ID. That is, the cracked client only get reassured by reassurance message that contain its name, and a message that reassure other boy does not help

6 Conclusion

Since crackers can track and analyze the application code, any client-side anti-piracy mechanism will only increase the cost of crackers, but will not be able to stop them in any real sense. The alternative is to put the anti-piracy logic on the server together with the core service of the software. However, this approach will harm the user experience. Intel's SGX technology allows us to put the anti-piracy logic in the client without compromising the user experience. This paper presents an anti-piracy framework based on the SGX extension of new Intel processors, using which applications can easily implement anti-piracy features.

Acknowledgment. This research work is supported by the National Natural Science Founds of China (62072368, U20B2050), Key Research and Development Program of Shaanxi Province (2021ZDLGY05–09, 2022CGKC-09).

References

1. Wójcik, B.: How to make cracker's life harder. Anti piracy protections for programmers. https://www.pelock.com/articles/how-to-make-crackers-life-harder-anti-piracy-protections-for-programmers (2019)
2. Cloosters, T., Rodler, M., Davi, L.: TeeRex: discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In: 29th USENIX Security Symposium (2020)

3. Wang, W., Liu, W., Chen, H., Wang, X., Tian, H., Lin, D.: Trust beyond border: lightweight, verifiable user isolation for protecting in-enclave services. IEEE Trans. Dependable Secure Comput. **20**, 522–538 (2021)
4. D'Agostino, B., Khan, O.: Seeds of SEED: characterizing enclavelevel parallelism in secure multicore processors. In: 2021 International Symposium on Secure and Private Execution Environment Design (SEED), pp. 203–209 (2021)
5. Youren, S., et al.: Occlum: secure and efficient muvltitasking inside a single enclave of Intel SGX. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (2020)
6. Gu, J.-Y., Li, H., He, Z.-Y.: Unified enclave abstraction and secure enclave migration on heterogeneous security architectures. J. Comput. Sci. Technol. **37**(2), 468–486 (2022)
7. Yavuz, T., Fowze, F., Hernandez, G., Bai, K.Y., Butler, K., Tian, D.J.: ENCIDER: detecting timing and cache side channels in SGX enclaves and cryptographic APIs. IEEE Trans. Dependable Secure Comput. **20**, 1577–1595 (2022)
8. Shweta, S., et al.: Binary compatibility for SGX enclaves. arXiv preprint arXiv:2009.01144 (2020)
9. Intel Software Guard Extensions (Intel SGX) SDK for Windows OS Developer Reference, Rev. 2.14.1 (2021)
10. Fei, S., Yan, Z., Ding, W., Xie, H.: Security vulnerabilities of SGX and countermeasures: a survey. ACM Comput. Surv. (CSUR) **54**(6), 1–36 (2021)
11. Zheng, W., et al.: A survey of Intel SGX and its applications. Front. Comp. Sci. **15**(3), 1–15 (2020). https://doi.org/10.1007/s11704-019-9096-y
12. Zhao, S., Li, M., Zhangyz, Y., Lin, Z.: vSGX: virtualizing SGX enclaves on AMD SEV. In: 2022 IEEE Symposium on Security and Privacy (SP), pp. 321–336. IEEE (2022)
13. Cui, J., Yu, J.Z., Shinde, S., Saxena, P., Cai, Z.: SmashEx: smashing SGX enclaves using exceptions. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 779–793 (2021)
14. Randmets, J.: An overview of vulnerabilities and mitigations of Intel SGX applications (2021). https://cyber.ee/research/reports/D-2-116-An-Overview-of-Vulnerabilities-and-Mitigations-of-Inte l-SGX-Applications.pdf
15. Wu, T.Y., Guo, X., Chen, Y.C., Kumari, S., Chen, C.M.: SGXAP: SGX-based authentication protocol in IoV-enabled fog computing. Symmetry **14**(7), 1393 (2022)
16. Chen, Z., Vasilakis, G., Murdock, K., Dean, E., Oswald, D., Garcia, F.D.: VoltPillager: hardware-based fault injection attacks against Intel {SGX} enclaves using the SVID voltage scaling interface. In: 30th USENIX Security Symposium, pp. 699–716 (2021)
17. Wei, W., Wang, J., Yan, Z., Ding, W.: EPMDroid: efficient and privacy-preserving malware detection based on SGX through data fusion. Inf. Fusion **82**, 43–57 (2022)
18. Liu, G., Yan, Z., Feng, W., Jing, X., Chen, Y., Atiquzzaman, M.: SeDID: an SGX-enabled decentralized intrusion detection framework for network trust evaluation. Inf. Fusion **70**, 100–114 (2021)
19. Kogler, A., Gruss, D., Schwarz, M.: Minefield: a software-only protection for SGX enclaves against DVFS attacks. In: USENIX Security Symposium (2022)
20. Kumar, S., Sarangi, S.R.: SecureFS: a secure file system for Intel SGX. In: 24th International Symposium on Research in Attacks, Intrusions and Defenses, pp. 91–102 (2021)
21. Nakano, T., Kourai, K.: Secure offloading of intrusion detection systems from VMs with Intel SGX. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pp. 297–303. IEEE (2021)

22. Yoon, H., Lee, M.: SGXDump: a repeatable code-reuse attack for extracting SGX enclave memory. Appl. Sci. **12**(15), 7655 (2022)
23. Toffalini, F., Graziano, M., Conti, M., Zhou, J.: SnakeGX: a sneaky attack against SGX enclaves. In: Sako, K., Tippenhauer, N.O. (eds.) ACNS 2021. LNCS, vol. 12726, pp. 333–362. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78372-3_13