Mobile App to SGX Enclave Secure Channel

Fernando Kaway Carvalho Ota
University of Luxembourg
Luxembourg, Luxembourg
fernando.carvalho@uni.lu

Jorge Augusto Meira
University of Luxembourg
Luxembourg, Luxembourg
jorge.meira@uni.lu

Cyril Renaud Cassagnes
University of Luxembourg
Luxembourg, Luxembourg
cyril.cassagnes@uni.lu

Radu State
University of Luxembourg
Luxembourg, Luxembourg
radu.state@uni.lu

***Abstract*—The current challenge for several applications is to guarantee the user's privacy when using personal data. The broader problem is to transfer and process the data without exposing the sensitive content to anyone, including the service provider(s). In this paper, we address this challenge by proposing a protocol to combine secure frameworks in order to exchange and process sensitive data, i.e. respecting user's privacy. Our contribution is a protocol to perform a secure exchange of data between a mobile application and a trusted execution environment. In our experiments we show independent implementations of our protocol using three different encryption modes (i.e., CBC, ECB, GCM encryption). Our results support the feasibility and importance of an end-to-end secure channel protocol.**

***Index Terms*—SGX, secure channel, mobile security, GDPR, trusted execution, data breach**

I. Introduction

Recurrently, events such as data breaches[1] put the user's privacy on the spotlight. At the business level, evolution of data protection regulations (e.g. GDPR in Europe) are pressuring service providers but there is a urge. The lack of enhanced secure user authentication, specially in banking environments, increases privacy concerns [1]. Some previous leakages of worldwide players, such as Facebook, show that the trade-off between security and convenience seems to always end in big losses, i.e. identify theft and confidentiality failure. If leaking personal conversations or pictures are considered big issues, the problem become even more critical when using sensitive personal data, such as biometric data. In the context of this paper, we focus exclusively on biometric data as a use case. Indeed, mobile banking is now a reality and so is the fingerprint reader embedded on most of the mobile phones. There is no doubt that biometric, implicit and continuous authentication [2], improve security of mobile applications. However it is paramount to take into consideration the threats that a potential exposition of this data might cause.

The main manufacturers of smartphones have implemented the extractor, matcher, database and decision system embedded in the operational system to enable collection of biometric data with the hardware sensor shipped within the device. In most of the cases, the embedded authentication method works fine. However, in case that a more complex process is required, the biometric data must be transferred from the device to an external server, which raises several concerns related to potential data breaches.

Research supported by EC H2020 Project CONCORDIA GA 830927

[1]https://www.privacyrights.org/data-breaches

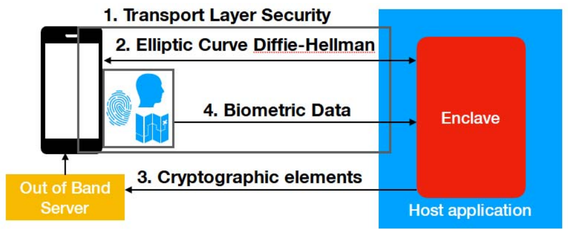
1. Transport Layer Security
2. Elliptic Curve Diffie-Hellman
4. Biometric Data
Enclave
3. Cryptographic elements
Out of Band Server
Host application

Fig. 1: Architecture overview to transfer and process data

In this paper, we propose a protocol to transfer sensitive data (e.g. biometric data) from a mobile device to a trusted execution environment, in which the data can be processed securely, see Figure 1. In this context, we list below those that we see as the main security concerns:

- Replay attack: An attacker with channel control between the sensor and feature extractor can resubmit a biometric template and gain new access;
- Bypassing the Extractor: If the attacker is controlling the feature extractor, it can modify any incoming biometrics;
- Synthesise vector of characteristics: by knowing how the biometric template is constructed, the attacker can synthesise the vectors to fool the *decision system.*

Furthermore, when it comes to biometric systems, we must consider vulnerabilities that could potentially be explored by adversarial machine learning [5] as the matcher is usually based on training models. What all these attacks have in common is that they are initiated from the data source because we consider, like most of today web applications, that the certificate authority of the service provider is not compromised. Indeed, certificate based mechanisms are usually provided to ensure that the requests are coming from an attested third party. However when they come directly from the mobile apps it is not possible to assure the origin of the requests. This mainly happens because the certificate, usually pinned in the app code, can be retrieved by a reverse engineering process. In possession of the certificate, an attacker can build his own app to send malicious requests to the bank API. With the enforcement of the Payments Services Directive 2 (PSD2), banks are opening their API to third party. This threat is serious because it impacts security measure on HTTP requests. PSD2 shall increase the HTTP requests coming from different third parties applications making more difficult to decide in a short time frame if the request is legitimate.

Then, if the data is processed in a non-reliable cloud environment, it is also necessary to protect the data from the provider in order to prevent unauthorized disclosure and tampering. Therefore, in order to provide a trusted execution environment, the Intel Software Guard Extensions (SGX) was chosen as the root of trust for the proposed protocol. In the server side, all operations are processed inside a SGX enclave to guarantee confidentiality and integrity.

The rest of this paper is organized as follows. Next Section presents the threat model that we considered to develop our protocol. Section III presents the architecture and describes the protocol itself (subsection III-C). In Section IV we discuss the implementation of the protocol and present results related to the overhead of using SGX. Section V shows the related work. In Section VI we discuss our ongoing future work. Finally, Section VII concludes the paper.

II. Threat Model

By pinning the certificates into the mobile app, the developers can assure that an original instance installation is communicating with the proper server. However, the server has no guarantee that incoming requests are from the original app. An attacker capable of modifying the source code can repack the app to redirect connections to a malicious server or even start to send forged requests directly to the server APIs. While distributing a fake app is an heavy attack requiring skilled engineers, sending direct requests to the APIs will only demand an attacker to extract the certificate from the mobile app. In the former scenario, securing the app distribution by hashing and signing the binary is possible but require user awareness. In the later scenario, once the attacker knows how to generate the credentials payloads, he can start to feed the system with fake biometric templates until the system starts to accept it as a original one. The attack surface can be seen in Figure 2[2]. Explicitly, our protocol mitigates attacks targeting the communication channel and the client app side, as mentioned above.

III. Architecture

Our main goal is to create a secure tunnel between the mobile app, more specifically the biometric sensor and the SGX enclave. As a SGX enclave can only be called by a host application, all the data coming from the mobile is forwarded to the enclave, but with an end-to-end encryption.

A. Software Guard Extensions (SGX)

Intel SGX isolates an execution environment by creating an abstraction called enclave with a protected physical memory. Enclave code and data have their access controlled by the CPU. Thus, a host application can create an enclave but cannot access its data unless explicitly passed through function variables. Developers need to separate the untrusted code from the one processed in the Trusted Computing Base (TCB). This property allows running code even when the environment is not fully trusted such as a cloud provider.

[2]This Figure is an adaptation from [6].

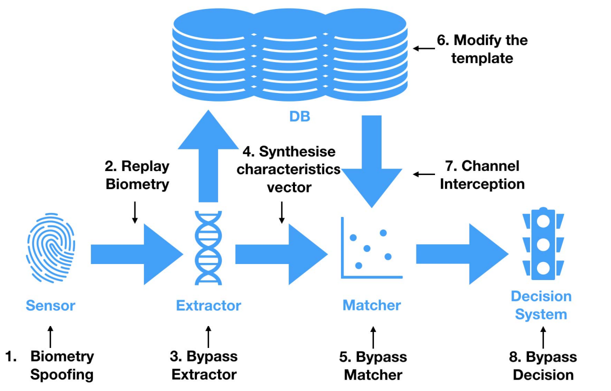

Fig. 2: Attack surface of a biometric system

Unfortunately, this benefit comes at a cost of memory and computation. The cache memory available to the TCB is 128 MB and part of it is used by itself, leaving only around 90 MB to the developers. The newer versions of SGX allows paging between cache and the main memory but to maintain the security, the pages are all encrypted, increasing the computation of the enclave code. A detailed explanation about SGX operation can be found in [14].

B. Properties

In order to cover the threats stated in the section II, the architecture offers the following properties:

1) Prevention to replay attacks: as the data leaves the application encrypted with an one time key, an attacker simply re-posting the message will not succeed.
2) Block upload of synthetic data (see Fig. 2 item 4): by closing a secure channel using an out-of-band server, it is unfeasible to insert synthetic data without tampering the mobile app and consequently modifying its signature.
3) Forward secrecy: even a future breach of SGX would block a Man in The Middle (MiTM) attacker of seeing previous data stored flowing through the channels or even inside the cloud provider.

C. Protocol

The Figure 3 presents a sequence diagram of the proposed protocol. A detailed description is provided below:

- Session cookie establishment (step 1): The protocol starts by establishing a TLS channel from the mobile app to the host app. This procedure should result in a session cookie, allowing the host app to identify the following requests coming from the mobile app. By pinning the host certificate in its code, the mobile app guarantee the communication with the known host.
- Mobile app and enclave key agreement (steps 3, 4, 5 and 8): A shared secret is reached by performing a Elliptic-curve DiffieHellman (ECDH) between the mobile app and directly the enclave by changing the public keys (PKm and PKe respectively). For auditing purposes or in case

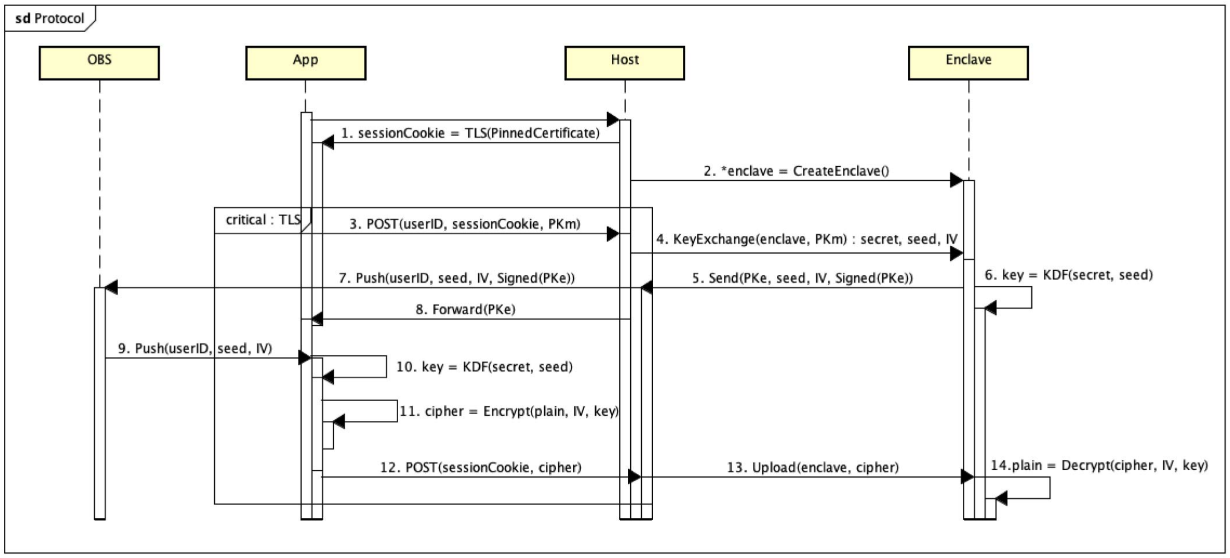
sd Protocol
OBS
App
Host
Enclave
1. sessionCookie = TLS(PinnedCertificate)
2. *enclave = CreateEnclave()
critical : TLS
3. POST(userID, sessionCookie, PKm)
4. KeyExchange(enclave, PKm) : secret, seed, IV
7. Push(userID, seed, IV, Signed(PKe))
5. Send(PKe, seed, IV, Signed(PKe))
6. key = KDF(secret, seed)
8. Forward(PKe)
9. Push(userID, seed, IV)
10. key = KDF(secret, seed)
11. cipher = Encrypt(plain, IV, key)
12. POST(sessionCookie, cipher)
13. Upload(enclave, cipher)
14.plain = Decrypt(cipher, IV, key)

Fig. 3: Protocol Sequence Diagram

the mobile app developer does not trust the host app provider, an enclave remote attestation can be included to certify that the mobile app is communicating with the right enclave.

- Out of band cryptographic elements for encryption mode (steps 5, 7 and 9): The enclave generated initialization vector (IV) and seed for the KDF are sent through an out of band server (OBS) along with their enclave signed signatures.
- Key Derivation Function (KDF) (steps 6 and 10) The key to be used for encryption shall be derived on both sides combining the shared secret agreed on ECDH and the seed coming from the OBS to generate the secret key (SK).
- Encryption of data (steps 11 to 14): This last step delivers the encrypted data to the enclave. For this purpose, the mobile app encrypts the data using symmetric encryption with the SK and the IV. The generated cipher is POSTed to the host app that uploads it to the enclave. The enclave is already in possession of the cryptographic elements necessary to decrypt.

In our implementation, we have used a hardcoded seed and IV (see Section IV for more details). Nevertheless, below we list potential out of band servers to be considered:

OTP devices: One Time Password would fit for generating the randomness to derive the cryptographic elements such as the seed for KDF. Unfortunately, this is already susceptible to social engineering, in special for bank apps, as it relies on the generated password that can be leaked by the user.

SMS: Cryptographic elements could be sent through SMS, nevertheless, countless attacks to this infrastructure have and are still being conducted all over the world as some financial institutions insist to send transaction authentication numbers (TAN) through this channel. The attacks include intercepting SMS at the Mobile Network Operator (MNO), device SMS leakage [11] and mobile phone line kidnapping. The latter consists on faking IDs of the owner and making the MNO to transfer the phone line to the fake customer, a very common attack in developing countries, such as Brazil.

ATM: Banking apps can count on their ATMs to agree on a first key or the first seed and/or IV. However, this would bind those cryptographic elements to an instance of the app installed on the device. If the user changes its device or even re-install the mobile app, then the process would need to be repeated. This reduces the user experience, but on the other hand provides high security mechanism.

Push notifications: In case of Android app, the IV can be sent through the Firebase push notification. As Firebase can validate the mobile app signature before delivering the message, an MiTM attacker would not receive the cryptographic elements necessary for the KDF and encryption steps. Therefore, the attacker will not be able to send data to the enclave and this implicitly guarantees that the enclave will receive it from the mobile app. In terms of performance, Google claims to deliver 95% of the messages within 250 ms. Apple users would also count on the same sort of mechanism.

IV. Experiments

In this section we present our experiments to assess the overhead of our SGX enclave implementation over three different encryption modes (i.e., CBC, ECB and GCM). On the server side, the experiments have been conducted on the Confidential Computing platform of Azure providing SGX support. An 3.7GHz Intel XEON E-2176G backs these machines along with 8GB RAM running on Ubuntu 16.04. On the client side, and Android app was created and installed on Google Pixel XL with Android 9. The measurements are averaged on 10 runs.

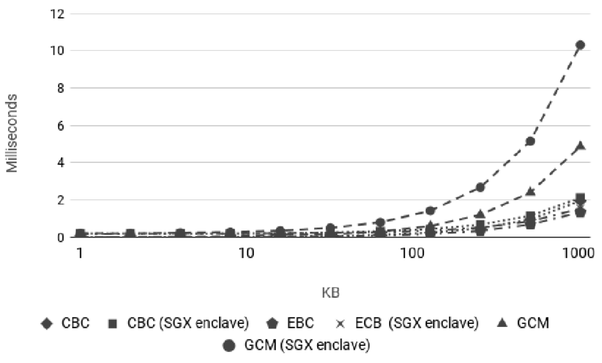

Fig. 4: Execution of CBC, EBC and GCM Encryption with and without SGX enclave implementation.

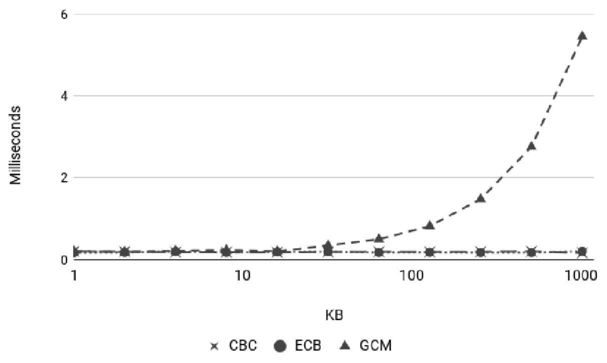

Fig. 5: SGX implementation overhead

A. The implementation

We have developed the host and enclave applications in C, using the Open Enclave framework. All the required cryptographic functions, namely ECDH, AES, ECB, CBC and GCM are included in the MBEDTLS library shipped with the framework.

In the host application a TLS 1.2 socket listens to the connection coming from the client. Once the TLS tunnel is created, the host application receives the first POST request containing a public key from the client to begin the ECDH. Then the host application starts the first servlet that creates the enclave and call the enclave key generation function. Upon completion of the function, the enclave public key is forwarded to the client. Meanwhile, the enclave completes its part of the ECDH and stores in its memory the secret. This secret is then used by the enclave when the host application calls the KDF, which consisting in applying SHA-512 8192 times along with a seed. For the tests we hardcoded the seed, however, that should go through the out-of-band channel along with the IV to be used in the encryption part. The source of random numbers for cryptographic elements shipped in the MBEDTLS version of the openenclave is insecure[3]. Instead, we used the function *sgx_read_rand* that calls SGX Random Number Generator (RNG).

After answering the first POST with the PKm, the host application calls the KDF function to generate and expand the secret and the seed to the 256 bits SK. This implementation order allows the client app to proceed with their part of the key exchange without waiting for the enclave KDF processing. The enclave is kept alive until the host application receives the second POST containing the cipher, so that the secret never leaves the enclave. Finally, the host application calls the decryption function with the cipher. For testing purposes, the AES encryption and decryption were implemented in operation modes ECB, CBC and GCM. In AES/GCM we implemented three versions of the function: static, optimized and dynamic. The static receives two arguments, the plain text and the cipher text with a fixed length of 1 MB, while the optimized only receives one buffer, reducing the TCB to at most this 1 MB less. The dynamic allocates memory to an unique buffer.

Some operations modes do not accept to use the same buffer for the plain and cipher texts, nevertheless, as the encryption and decryption are applied for each 16 bytes, the unique buffer passed from the host application to the enclave is replaced block by block. By using the Native Development Kit (NDK) provided by Android, the client app implemented the same cryptographic functions in C from the MBEDTLS. Both POST messages (Key Exchange and Cipher Upload) were implemented as asynchronous tasks to allow the Android app to call them upon demand. In the mobile side, we used the SecureRandom class with a strong RNG provided by Android to generate the ECDH seed[4].

B. Results

The figure 4 shows performances for encryption in the three operation modes of AES in the enclave and directly in the host application. It is possible to see that the SGX implementation always impose an overhead to the execution time. The overhead is linear for ECB and CBC cipher modes as seen in figure 5, but not for GCM. We tested up to 1 MB as our use case is biometric templates and only for encryption as we have not seen any major differences for decryption. In any case, the overhead in encryption was never bigger than 6 ms per MB.

We also tested the overhead for creating the enclave, key exchange and a Key Derivation Function (KDF). The results presented in Figure 6 can be all taken into account when using a SGX enclave to do a ECDH to reach a cryptographic key. In practical terms the KDF time is negligible on the server side, as its computation can be processed while waiting for the second POST. The key exchange function (item 4 of Fig. 3), namely, generating the random seed and calculating the shared secret using the public key from the mobile, has also a

[3]https://github.com/microsoft/openenclave/blob/master/3rdparty/mbedtls/mbedtls/yotta/data/README.md

[4]https://developer.android.com/reference/java/security/SecureRandom

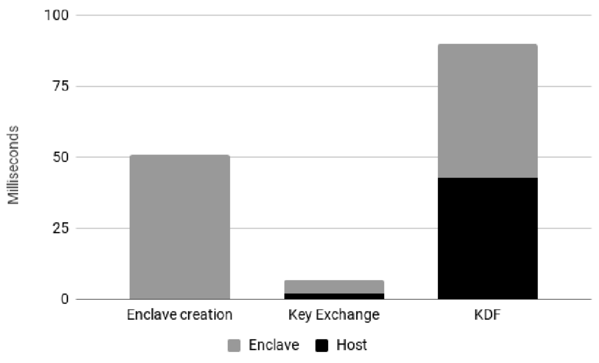

Fig. 6: Key exchange overhead

negligible overhead of less than 5 ms. The only considerable overhead is the enclave creation that takes an average of 50 ms, which can not be considered an issue for practical use.

C. Discussion

In this section we discuss in details the potential vulnerabilities caused by compromised components in our proposed architecture.

The host application added enclave life-cycle functions to the servlet with TLS of the host application implemented in C. This first implementation of the servlet maintain the enclave state between key exchange and data upload requests. In order to make it stateless, an enclave binded function can seal the key and retrieve it when the ciphered data comes. However, this can lead to break the forward secrecy in case of an attacker saving the history in possession of a compromised sealing key.

The three cipher modes tested are common in mobile apps. The assumption was that a more complex cipher mode would increase the overhead of using SGX. However, the results showed that only the GCM would increase the overhead exponentially. By all means it is highly recommended to use a Authenticated Encryption with Associated Data (AEAD) mode provided by GCM over CBC [10] if the amount of data allows as GCM tends to rocket for large files. In comparison to the insecure ECB mode [10], the enclave overhead of using CBC is quite small even if the file increases too much. Therefore, developers should not choose to implement ECB instead CBC in any situation of this protocol as there are no considerable performance penalties. In addition, it is important to consider the possibility of each of the components present in the entire process being compromised. To this analysis, we consider the most secure scenario using AES in GCM mode and the OBS as a push notification service.

- Compromised Mobile App: any tampering in mobile application code would modify its signature, disallowing the mobile application to request an user push notification ID. Therefore, by only attacking the mobile app, the attacker will not be able to deliver a message containing the seed and IV (item 9 - Figure 3) to an original instance of the app. So when the attacker build the cipher, the enclave will not decipher it correctly, the tag checking method of GCM would fail and the message can be safely discarded. Additionally, to act as MITM, the attacker would need to modify the pinned certificate in the code and redirect the connection to his server, consequently modifying the signature as well. Moreover, the distribution of the modified app would require social engineering techniques.
- Compromised Device: jailbreak or rooted devices poses as a challenge to major mobile apps. However in this case, it is not only a matter of owning the device but also the push notification services built in the operational systems of the Android and iOS. This would allow the attacker to retrieve the seed and IV for the encryption. Old versions of those services had abuses reported [13], but the new ones apparently remain intact.
- Compromised Host application: this piece of software is a natural MITM in the environment as forwards all the messages between all the other players. Yet, if a key not generated by the enclave is sent by the host application during step 8 (see Figure 3), its signature cannot be forged if the mobile app is remotely attesting the enclave sign key, preventing the creation of the tunnel with the mobile app. This is efficient in guaranteeing the privacy of user's data. Unfortunately, it does not protect from the host application from uploading fake data to the enclave since it can exchange keys directly with the enclave. This can be mitigated by having the enclave checking the signature of the host app.
- Compromised Enclave: the enclave is the root of trust. In other words, if the enclave is corrupted during runtime, the whole setup fails. However, if a breach that allows to recover the keys of SGX is founded, the previous communications are not compromised as the shared key that encrypts sensitive data is immediately discarded after use upon enclave destruction.
- Compromised Out of Band Server: an attacker controlling the OBS can deny delivery of the messages and make the services unavailable.

Only by exploiting combined different infrastructures, that are nowadays reliable, an attack would succeed. For instance, if external attacker objective is uploading fake/synthetic data to the enclave, he would need to modify the mobile application, rooting/jailbreaking the device and bypassing push notification services of main mobile operational systems, which has not been done in the latest versions of the latter. In case the attacker is targeting to intercept data from the users, he would require to thrive in all those aforementioned three attacks inside the target user's device. In short, those attacks are unfeasible in practice using the current technology of the components we implemented. Depending on the specifications of the Hardware Secured Model (HSM), the latter can replace the enclave, in other words, our protocol can be extended to current major banking infrastructure.

V. Related work

The use of trusted zones, such as SGX enclaves, to process sensitive data is relatively new and the number of real use cases are not numerous. Researchers are now starting to put more efforts on this subject, driven by the recent threats and data breaches, as mentioned in session I and II. In the work presented in [4], the authors focus on analysing the biometric data from mobile devices by using enclaves. They implement and evaluate three SGX-compatible learning algorithms: Naive Bayes (NB), k-Nearest Neighbour (kNN) and Logistic Regression (LR). While the authors of this work concentrate their efforts on analysing the data safely, we focus on presenting a secure way of delivering the data from the mobile to the enclave. Then, in [7], the authors present a mechanism to defend MITC attacks by using SGX. They use sealing and attestation features to protect the user credentials. Nonetheless, this solution considers a self-owned cloud server, in private cloud environment. Next, in [8], the authors present a implicit authentication system. They propose a profile matching function by using statistics of features to accept or reject a new sample presented by a user. The data is encrypted and stored at the carrier to avoid data leaks on the mobile side. After that, in [9], a software abstraction approach is proposed to offer trusted sensors to mobile applications. In fact, the aim is to give to the mobile application means to verify the authenticity of the data produced by the sensor. In short, we can argue that the related work, mainly [4], [8], [9], is complementary to ours, once we deal with the same problem but from a different perspective.

VI. Future Work

The code produced for this paper is not currently available. We are preparing a public version of our framework to be pushed to a public repository (e.g. github). In our ongoing future work we aim to extend our framework by considering the following aspects:

- Investigating compromised systems: we aim to investigate in more details each of the aspects presented in section IV-C, e.g. malicious host application. All the potential vulnerabilities will be attempted to break in order to see if a real attacker would succeed.
- Enable template extraction inside enclave, mitigating extractor bypassing (Fig. 1 item 3): the enclave can receive raw data from the sensors. This means that the extractor can reside protected inside of the enclave, as well as tailored for the needs of the analysis.
- On demand parser: after testing the security and building the template extractor, the next goal is to create a parser with a Domain Specific Language (DSL) to remove the Personal Identifiable Information (PII) incoming from the mobile to the enclave. The DSL should describe to the enclave which PII is going to be deleted so the data can be sent to analysis preserving privacy.

VII. Conclusion

In this work we proposed a protocol to transfer and process sensitive data in a trusted zone, such as SGX enclave. Our main target is to provide a secure channel to transfer data from a mobile device to a server in which the data can be safely analysed without exposing Personal Identifiable Information not even to the service provider. We proceeded experiments over three implementations of our protocol using three different encryption modes (i.e., CBC, ECB, GCM encryption). The target is to measure the overhead of using the SGX to create the encryption key for the secure channel. In general the performance impact of key generation is considerably low. Moreover, in our ongoing future work, we are focusing on the security side of our approach. We aim to evaluate the system under different attacks, taking into consideration the possibility of each of the components present in the entire process being compromised.

References

[1] Drake, C., & Gauravaram, P. (2019). "Designing a User-Experience-First, Privacy-Respectful, High-Security Mutual-Multifactor Authentication Solution". Security in Computing and Communications, 183-210.

[2] Alizadeh, Mojtaba, et al. "Authentication in mobile cloud computing: A survey." Journal of Network and Computer Applications 61 (2016): 59-80.

[3] Akhtar, Z., Micheloni, C., Piciarelli, C., & Foresti, G. L. (2014). "MoBio_LivDet: Mobile biometric liveness detection". 2014 11th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS).

[4] Shepherd, Carlton, Raja Naeem Akram, and Konstantinos Markantonakis. "Towards trusted execution of multi-modal continuous authentication schemes." Proceedings of the Symposium on Applied Computing. ACM, 2017.

[5] Huang, Ling, et al. "Adversarial machine learning." Proceedings of the 4th ACM workshop on Security and artificial intelligence. ACM, 2011.

[6] Akhtar, Z., Micheloni, C., & Foresti, G. L. (2015). "Biometric Liveness Detection: Challenges and Research Opportunities". IEEE Security & Privacy, 13(5), 6372. doi:10.1109/msp.2015.116.

[7] Liang, Xueping and Shetty, Sachin and Zhang, Lingchen and Kamhoua, Charles and Kwiat, Kevin. "Man in the Cloud (MITC) Defender: SGX-Based User Credential Protection for Synchronization Applications in Cloud Computing Platform". CLOUD 2017

[8] Nashad Ahmed Safa and Reihaneh Safavi-Naini and Siamak Fayyaz Shahandashti. "Privacy-Preserving Implicit Authentication" . IACR Cryptology ePrint Archive 2014.

[9] Liu, He & Saroiu, Stefan & Wolman, Alec & Raj, Himanshu. (2012). "Software abstractions for trusted sensors" . 10.1145/2307636.2307670.

[10] Rogaway, Phillip. "Evaluation of some blockcipher modes of operation." Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan (2011).

[11] Mulliner, Collin, Nico Golde, and Jean-Pierre Seifert. "SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale." USENIX Security Symposium. Vol. 168. 2011.ndthebibliography

[12] Firdaus, A., Anuar, N. B., Razak, M. F. A., Hashem, I. A. T., Bachok, S., & Sangaiah, A. K. (2018). Root Exploit Detection and Features Optimization: Mobile Device and Blockchain Based Medical Data Management. Journal of Medical Systems, 42(6). doi:10.1007/s10916-018-0966-x

[13] Esposito, C., Palmieri, F., & Choo, K.-K. R. (2018). Cloud Message Queueing and Notification: Challenges and Opportunities. IEEE Cloud Computing, 5(2), 1116. doi:10.1109/mcc.2018.022171662.

[14] Costan, Victor, and Srinivas Devadas. "Intel SGX Explained." IACR Cryptology ePrint Archive 2016.086 (2016): 1-118.