Trusted Inter-Process Communication Using Hardware Enclaves

Newton C. Will
Computer Science Department
Federal University of Technology - Paraná
Dois Vizinhos, Brazil
will@utfpr.edu.br

Tiago Heinrich, Amanda B. Viescinski, Carlos A. Maziero
Computer Science Department
Federal University of Paraná State
Curitiba, Brazil
{theinrich, abviescinski, maziero}@inf.ufpr.br

***Abstract*—*Inter-Process Communication (IPC)* enables applications to share information in a local or distributed environment, allowing them to communicate with each other in a coordinated manner. In modern systems this mechanism is extremely important, as even local applications can run parallel tasks in multiple processes in the machine, needing to exchange information to coordinate their execution, and optimizing the exchange of data in a more efficient way. The security in IPC relies on the integrity and confidentiality of the messages exchanged in such an environment, as messages exchanged between different processes can be targeted by attacks that seek to obtain sensitive data or to manipulate the application behavior. A *Trusted Execution Environment (TEE)* can be used to enable an isolated execution of the IPC mechanism to mitigate such attacks. In this paper we propose the adoption of the Intel *Software Guard Extensions (SGX)* architecture to provide data confidentiality and integrity in message exchange between processes, by using hardware instructions and primitives to encrypt and authenticate the messages. Our approach highlights a threat model and compares the solution proposed with two other scenarios, showing a feasible solution for security and an approach that can be applied to standard IPC mechanisms with small processing overhead.**

***Index Terms*—Computer security, Data security, Information exchange, Operating systems and Software Guard Extensions.**

I. Introduction

The environment found in today's systems may be complex, with multiple processes sharing system resources and interacting among them to exchange information and to coordinate their activities. Usually the *Operating System (OS)* provides *Inter-Process Communication (IPC)* mechanisms that allow processes to exchange messages and perform synchronization [1, 2].

An IPC could explore distinct forms of communication that consider the purpose of the application [3], where multiples process or just two processes are communicating with each other; also, restrictions may exist for reading/writing in the communication channel [4].

Like any other software system, the OS IPC mechanisms are prone to security problems. In particular, the integrity and confidentiality of data exchanged through IPC is a main concern. An attacker could exploit vulnerabilities in an IPC mechanism to build attacks against the system, as for example a *Man-In-The-Middle (MITM)* attack [5].

This paper explores using the Intel *Software Guard Extensions (SGX)* to improve the security of an IPC mechanism. Intel SGX provide trusted execution environments in which security-critical code and data can be protected from the running environment. Using SGX enclaves, the exchanged messages get a new layer of security, ensuring data confidentiality and integrity. We apply the SGX attestation protocol and make use of the isolation primitives provided by SGX to encrypt and authenticate the message payload, improving the security of process communication.

This paper is structured as follows: Section II presents the background concepts for understanding of the paper. Section III presents the related work. Section IV presents our proposal. Sections V and VI present the evaluation results and discusses the security solutions found. Finally, Section VII concludes the paper.

II. Background

This Section presents a brief description of important concepts used in this work, such as *Inter-Process Communication (IPC)* and Intel *Software Guard Extensions (SGX)*.

A. Inter-Process Communication

Currently, complex software systems are structured as multiple processes running, sharing resources, cooperating, and coordinating their activities. The mechanisms that allow process to interact, exchanging information and synchronizing their executions are collectively called *Inter-Process Communication (IPC)*; there are several mechanisms available to achieve it.

Usual IPC mechanisms include *UNIX Pipes*, which are unidirectional stream-based bounded data channels between pairs of processes; *Shared Memory*, which allows multiple processes to access the same region of memory to exchange data efficiently but provides no coordination; *Message Queues*, providing a bounded buffer for messages that can be accessed by multiple processes for sending and receiving messages with strong synchronization constraints; and so on [1, 2, 4]. Synchronization mechanisms like *semaphores* and *condition variables* are considered IPC as well, as they provide means for tasks to interact.

A *Message Bus* is a specific IPC mechanism that routes messages among processes in a system. Each process connects itself to the bus using a name or an ID that can be used by the other processes to send messages to it. A typical message bus can also be used to broadcast messages to all or a subset of tasks, and supports the publisher/subscriber style of communication. D-Bus [6] is a message bus used in modern Linux systems, aiming at replacing the chaos that multiple processes would have exchanging messages between each other by a common path for all messages.

B. Intel Software Guard Extensions

In the last years some hardware mechanisms are being proposed to support the *Operating System (OS)* and applications to ensure the safe execution and manipulation of sensitive user data. One of such technologies is Intel *Software Guard Extensions (SGX)*, which allows the application to be divided into two components: *trusted* and *untrusted* [7]. The trusted component, called *enclave*, is placed in a protected memory region, called *Processor Reserved Memory (PRM)*, that is fully encrypted, with the encryption key handled only by the *Central Processing Unit (CPU)* and generated on each boot.

The access to the enclave is protected by SGX hardware, preventing enclave data from being accessed by malicious software or any software with high privileges, such as the OS kernel, virtual machine monitors and the BIOS, as shown in Fig. 1 [8]. Only the application that creates the enclave can access it, using public entry points called the *ECALLs*. Each enclave must have at least one public ECALL, allowing the application to access the enclave resources in a controlled manner. The enclave runs on ring-3 and cannot access system calls directly, requiring the definition of a set of functions that allow it to access the application untrusted code, the OCALLs [7].

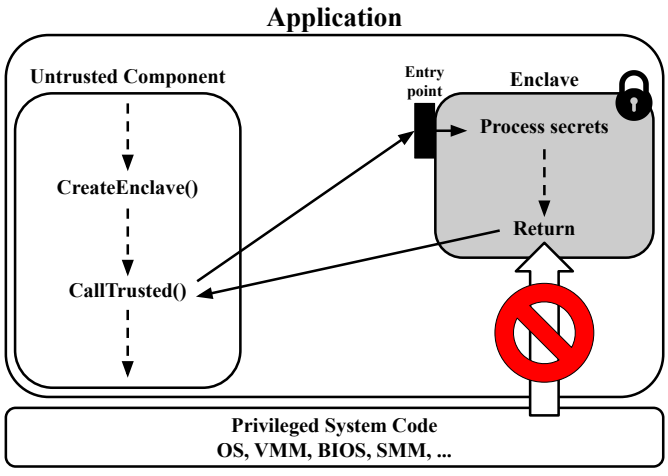

Fig. 1. Intel SGX operating principles [9].

Since all resources are statically linked to the enclave, a signature is generated at compile time containing information about the code and data to be placed in the enclave, including version and identification of the enclave author. This signature is validated when the enclave is loaded into the memory, ensuring that it is legitimate and has not been tampered with. After the enclave is loaded, CPU memory protection mechanisms validate each access to enclave data and code [8].

Finally, the SGX architecture provides mechanisms to keep the data confidentiality and integrity when stored on secondary memory. The enclave can use an encryption key derived from its signature and CPU to encrypt and authenticate the data. This key does not need to be stored by the enclave, and can be requested at any time. Also, enclaves can create a trusted channel to communicate each other locally or remotely, ensuring the confidentiality and integrity of data being exchanged [10].

III. Related Work

Secure communication among processes is well explored in microkernel architectures, by using virtual address space switching [11, 12] and hardware-based security mechanisms, such as the Intel *Memory Protection Key for Userspace* [13]. In monolithic kernels, a set of works focuses on moving the IPC mechanism to the kernel level, in order to achieve better performance [14, 15].

Another approach seeks to provide secure and transparent communication between applications and the OS [16]. The authors propose a reliable communication layer located between the kernel and the user space. To provide authenticity and integrity to communications, an isolated memory space is reserved for them and the messages are protected by cryptographic hashes. In addition, all interactions between processes are monitored and controlled by access control policies enforced by a reference monitor. A prototype was implemented and evaluated to validate the proposal; the results presented showed a high overhead during communications.

The study [3] raises a discussion about the security features that the operating system must support to contribute to secure communication between local processes, based mainly on SELinux. It proposes a unidirectional IPC mechanism using shared memory combined with message queues to perform data forwarding, but no evaluation of the proposal implementation is presented, impairing its validation.

CurveCP [17] focus on the security in transmitting packets in an open network, where common network protocols sometimes do not implement the appropriate security measures. Focusing on fixing problems concerning the integrity and confidentiality of the data transmitted, CurveCP relies on a connection with similarities with *Transmission Control Protocol (TCP)*. CurveZMQ is an adaptation of CurveCP that expands it for protocols like *User Datagram Protocol (UDP)*, using an elliptic curve algorithm with a key of 256 bits for client/server communication. This way, each new session a pair of keys is stored in memory during the communication interval, fixing problems with stolen keys and MITM attacks [18].

Finally, [19] builds a library that ensures secure communication between process by using SGX mechanisms. This library is placed over the *libdbus* default implementation and can be used to mediate the communication between the application

and the message bus, providing attestation and encryption features. The solution presents a high overhead and its limited only to basic data types.

IV. A Trusted IPC Approach

As described in Section II, IPC allows different applications communicate each other and there are distinct ways to implement IPC mechanisms. Message buses are largely used as IPC mechanisms, allowing applications to send and accept messages using one or more buses (see Fig. 2), allowing message exchange without knowing implementation or protocol details about each other. To send a message, the application just puts it on the bus, and to receive a message, it only needs to listen the bus and collect the messages addressed to it.

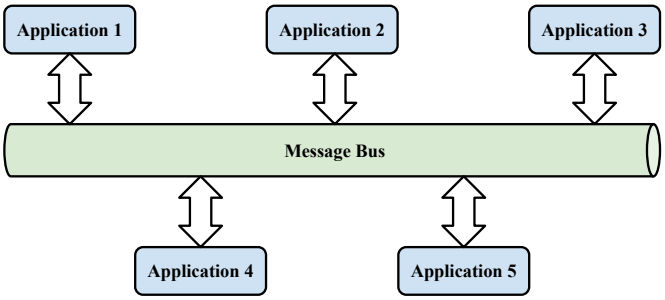

Fig. 2. Message bus architecture.

This IPC architecture presents several advantages, allowing adding and removing applications from the bus with no impact on other applications connected to it, with the applications working in a decoupled fashion. The communication complexity is reduced in the application, which only needs to follow the bus protocol, letting the bus handle all routing and interconnection complexity. This way, the applications only needs to know how to communicate with the bus; also internal bus components can be changed to meet new requirements without impact the applications. All these characteristics made message buses highly modifiable and scalable [20].

On the other hand, the bus become a single point of failure for all connected applications, and can be exploited to interfere with communications and violate message integrity. Also, message buses offer no privacy nor confidentiality, with the messages delivered indiscriminately to all applications connected to the bus, each application being responsible for filtering messages addressed to it.

Despite these security limitations, some applications use message buses to transmit sensitive data, such as user credentials, applying cryptographic algorithms to ensure the confidentiality of exchange data [21]. Such solutions do not ensure integrity nor provide message authentication, being susceptible to active MITM attacks. A wide range of applications, even using other IPC mechanisms, present leakage of sensitive data that may compromise the security of the application and the privacy of users [22].

In this context, we propose the use of SGX enclaves to create trusted channels inside open message buses, providing a secure way to exchange sensitive data between applications. Our solution ensures data confidentiality and integrity by using the trusted execution environment and root of trust provided by the SGX architecture.

A. Bus Connection

In a message bus, each application must connect to the bus to be able to send messages to other applications that are connected to it. In this step, the application receives an *ID* that uniquely identifies the connection. Each application can establish multiple connections to the bus, each one with a different *ID*. The application can be found by others by using this connection *ID* or, alternatively, a registered alias.

Our solution creates an SGX enclave at this step, which will be uniquely linked to the connection. In this way, each connection will manage its own enclave. This enclave will provide a trusted execution environment to perform the message encryption/decryption and handle the encryption keys. Also, the enclave provides a root of trust that allows to authenticate the key agreement procedure and all exchanged messages.

B. Creating a Trusted Channel

Once the application is connected to the bus, it can send messages to other ones that are also connected to the same bus, identifying the receiver by its ID or alias. The application can also listen the bus and read all messages, filtering which ones are addressed to it. This is one of the main characteristics of a message bus: each application connected to the bus can listen and read all messages put on the bus, even when the message was not addressed to it.

Thus, to allow sending confidential data over the bus, our solution enables the creation of trusted channels, ensuring a secure communication between two applications (Fig. 3). The trusted channel is created by using the SGX local attestation procedure.

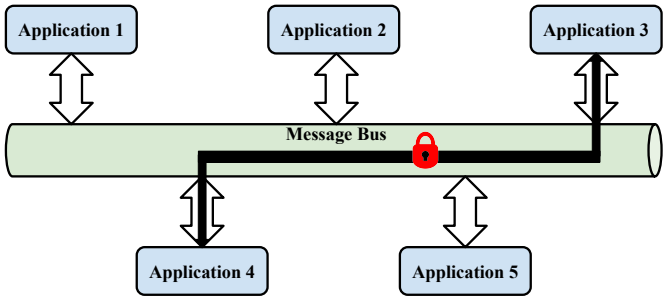

Fig. 3. Trusted communication channel in a message bus.

SGX local attestation allows an enclave to be confident that another enclave is running in the same platform and was properly instantiated. To build this proof, the enclave asks to the SGX hardware to build a credential, called *REPORT*, that reflects the enclave signature, including information about its

security properties. This report can be provided to another enclave, enabling it to verify the identity of the sender [10].

The attestation procedure has three steps, as shown in Fig. 4. ① Enclave A obtains the identity of enclave B; ② Enclave A creates the *REPORT* structure using the identity of enclave B and sends this structure to enclave B; ③ Enclave B uses the data contained in *REPORT* sent by A to verify that enclave A is running on the same device as B; enclave B may create a *REPORT* structure and send it back to enclave A, which will perform the same validation. All these steps can be performed over an open communication channel and, at the end, the enclaves will be mutually authenticated. Also, both enclaves use the exchanged information to perform an authenticated key agreement.

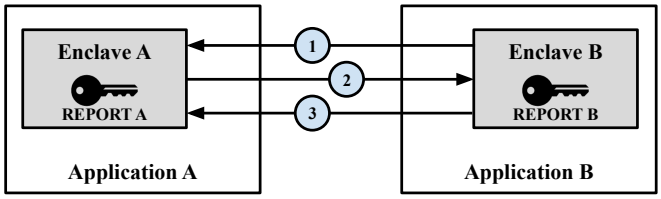

Fig. 4. SGX local attestation procedure [10].

After the attestation process, the two applications have a trusted communication channel to exchange sensitive messages in a secure way over the bus. The same bus connection can handle multiple trusted channels with different applications. The trusted channel can be closed or reset at any time by any of the applications involved.

C. Secure Messaging

The attestation process provides an 128-bit key as a result of an authenticated Diffie-Hellman key agreement. This key is stored by both enclaves and can be used to encrypt sensitive data before putting them on the bus. To do this, our solution uses the enclave to perform the data encryption, ensuring that the encryption key never leaves the enclave boundaries, keeping it protected by the SGX mechanism.

Since SGX provides a root of trust, the encryption is performed using an authenticated AES-GCM algorithm, including a message signature that is used to identity whether any part of the message was tampered with. Thus, we ensure the message integrity, identifying corrupted data and notifying the sender.

To ensure message confidentiality, all message payloads are encrypted, including the message signature, which identifies what kind of data is being sent. Message metadata, such as message type, sender and destination are not encrypted, since they must be readable by all applications connected to the bus to be filtered by each one.

V. Performance Evaluation

To evaluate our proposal, we implemented a proof of concept based on the D-Bus message bus [6], which is the main IPC mechanism used in current Linux desktops. D-Bus uses a daemon to enable the connection of applications to the bus and perform the message routing; and a low-level *Application Programming Interface (API)*, called *libdbus*, that provides the protocol implementation and allows the applications to exchange messages among them.

Our prototype adds new functions to the *libdbus* to allow applications to create trusted channels on the bus. This section presents the tests that were performed to measure the overhead imposed by our implementation.

A. Experimental Setup

We run the performance tests in a Dell Inspiron 7460 laptop with the following settings: dual-core 2.7 GHz Intel Core i7-7500U CPU, 16 GB RAM, 1 TB hard-drive, 128 GB SSD, SGX enabled with 128 MB PRM size. Intel *TurboBoost*, *SpeedStep*, and *HyperThread* extensions were disabled, to provide stable results. We use Ubuntu 18.04 LTS, kernel 4.15.0-112-generic, *libdbus* 1.12.20, Intel SGX SDK 2.9.101.2, and set the enclave stack size at 8 KB and the enclave heap size at 64 KB.

B. Methodology

To evaluate the method proposed, three distinct test scenarios were elaborated, with the focus on reflecting common operations between a client and a server process:

- **Scenario 1:** client connect to the bus and send a message addressed to server, with no encryption;
- **Scenario 2:** client connect to the bus and perform a key agreement with the server by using an *Elliptic-Curve Diffie–Hellman (ECDH)* protocol based on Curve25519 [23]. The key is then used to encrypt the message using a 128-bit AES-CTR algorithm;
- **Scenario 3:** presents our solution. Client connect to the bus and creates a trusted channel to communicate with the server.

We also measured the overhead of our solution in three aspects: bus connection, payload encryption, and message throughput. All benchmarks were taken by using the RDTSCP instruction, which provides a high-precision low-overhead time source [24]. Experiments were run 10,000 times each. The width of the confidence interval at 95% is 1.55% of the average.

C. Connection and Session Establishment

The connection to the bus is the first step that the application must perform in order to be able to send messages to other ones, as described in Section IV-A. In this step, we create an enclave that will be bound to the connection. Also, to establish a trusted channel, the application must perform the attestation procedure (Section IV-B). Both operations impose a high overhead, as shown in Fig. 5.

The overhead is mainly caused by enclave creation, more than 70% of total time spent, due to the need for the enclave to allocate all the necessary memory in PRM at the creation time. This operation has a high computational cost, as described by [21] and [25]. It is important to highlight that, once the enclave is created, it will be able to handle several trusted communication sessions. This factor can be considered as a

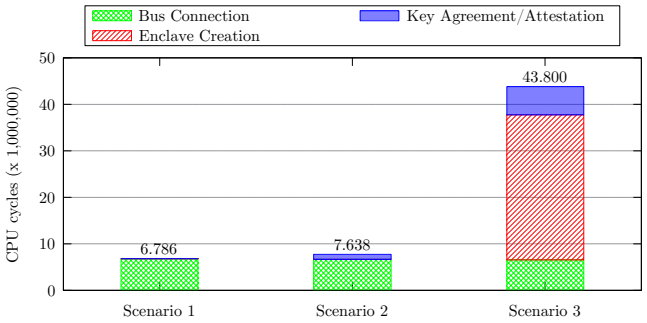

Fig. 5. Connection/session establishment overhead.

trade-off between a secure system and an insecure one and could be better explored in systems where threads are already mapped and optimization could be made [26].

Finally, the attestation step also has a considerable impact, having a computational cost of approximately 6×, compared to the ECDH key agreement. This overhead is the result of the security mechanisms used to authenticate and validate the data exchanged between enclaves at this step, which provide a root of trust and allow to use the agreed key to authenticate all data exchange between them after the attestation.

D. Payload Encryption

After the key agreement step, both applications can send encrypted messages to each other. The encryption adds an overhead in message sending, as it features an extra step in the procedure. To measure this impact, we evaluate the time spent when sending messages with different payload sizes, from writing the message by the client to receiving and reading it by the server. The results are presented in Fig. 6.

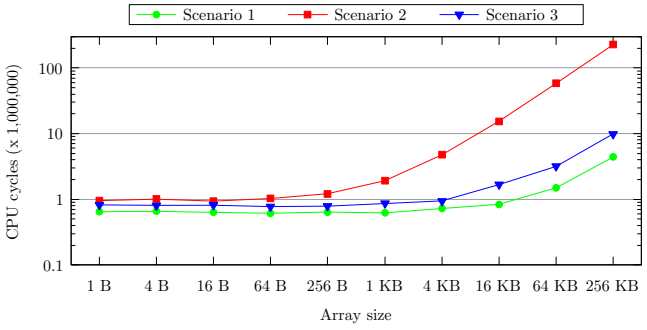

Fig. 6. Overhead encrypting messages with data array payload.

We can see that the time spent to send a message increases as the payload size grows, as well as the encryption overhead, and we could say that is an expected behavior considering others studies [19, 26, 27]. Scenario 3 presented a better performance, with an overhead of approximately 28% for a message payload up to 4 KB, with respect to scenario 1, and has 18% less impact when compared to scenario 2.

Considering messages with a payload bigger than 4 KB, scenario 3 presents an overhead of approximately 100% when compared to scenario 1, but still with a performance superior to that presented in scenario 2.

E. Message Throughput

Aiming to know the amount of messages and data the bus can handle in short periods of time, we use a "ping-pong" application in which the client sends a message, the server reads it and sends the same message back to the client. Fig. 7 presents the number of messages dispatched per second; we can see that scenario 3 presents a throughput around 30% lower than scenario 1 for messages with payload up to 16 KB. Above this threshold, this difference is reduced to 15%. Scenario 3 also presents a better throughput than scenario 2, which has a performance drop starting at a payload size of 64 KB.

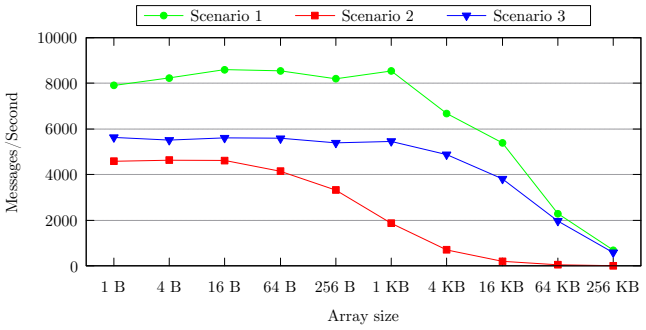

Fig. 7. Message throughput with different payload sizes.

We can see that the number of messages dispatched drops when payload size increases, even in scenario 1. Scenario 2, which presents a high encryption overhead, has also a perceptive drop in message dispatch rate starting at 64 Bytes of payload size, earlier than presented by the other scenarios. Scenario 3 presents a smoother drop, achieving results close to the scenario 1 to messages with payload equal or higher than 64 KB.

To complement the results presented in Fig. 7, we evaluated the amount of data sent per second, with results shown in Fig. 8. The results show that scenario 2 is limited to a throughput of 3.5 MB/s, while scenarios 1 and 3 present transfer rates of 175 MB/s and 150 MB/s, respectively. These results demonstrate that our solution presents scalability with a low computational cost.

VI. Security Assessment

This section presents the threat model and the security analysis of our solution.

A. Threat Model

We consider a threat model where the adversaries aim to have access to the data that travels on the bus. To do this, they can listen the bus and record all data retrieved from it or send such data to another machine over the network, maybe in real time. By sending the messages to another machine, they can read messages with plaintext payload and use higher

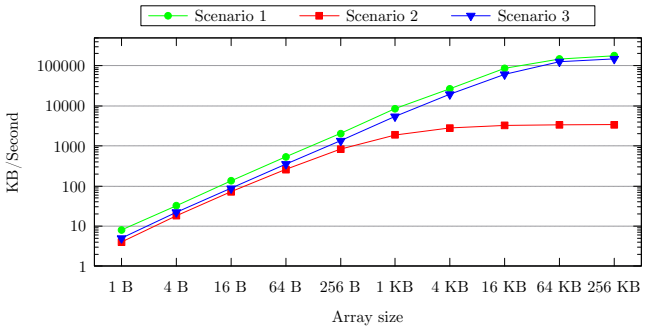

Fig. 8. Byte throughput with different payload sizes.

computational power to apply techniques to break payload encryption.

Attackers can also intercept and replace the messages, performing an active MITM attack aiming to replace the messages placed on the bus or interfering with a key agreement between two applications. Finally, they can also perform a memory dump or a *Direct Memory Access (DMA)* operation to extract the encryption keys defined by the applications when using a key agreement protocol.

B. Discussion

To assess the security of our solution, we consider that the Intel SGX architecture works properly, according to its specifications, and it is secure, focusing our validation only on the proposed solution. We also consider that the development environment is reliable.

If the attacker has access to the machine, or can manage to install a malicious software to listen the bus, they will get only the encrypted data, since the payload is encrypted before being put on the bus and can only be decrypted by the receiving application, which performed the attestation and key agreement with the sender. Even if the attacker stores the collected messages to break the encryption key in a high performance environment using a brute force strategy, this will take an excessively long time, making it impractical.

Even when acting actively on the bus, replacing the messages sent on it, the attacker will not be successful. At the attestation an key agreement step, any interference will be detected by the SGX architecture, since the exchanged data reflect the unique signature of the enclave and is authenticated by the hardware. Thus, in an attempt to manipulate the attestation process, it will be interrupted and both applications will be notified. The replacement of encrypted messages after the establishment of the trusted communication channel will also be detected, since it is authenticated by the SGX hardware. In this case, the receiver will notify the sender that the message was corrupted.

Finally, if the attacker uses a malicious software or similar techniques to perform a memory dump or DMA operations in order to obtain the keys used to encrypt the messages, they will not have success, since the keys are kept inside the enclaves and never go out their boundaries. The enclave memory is encrypted with a 128-bit AES-CTR algorithm, with the encryption key being randomly generated on each boot and stored on CPU registers, and the enclave data and code are decrypted only inside the CPU.

VII. Conclusion

This paper presented a novel solution to provide a trusted communication channel over a message bus, by using software isolation and protection mechanisms provided by Intel SGX architecture. We implemented a proof of concept by changing the *libdbus* low-level API to validate our proposal.

The performance evaluation shows a small overhead on data encryption and decryption, when compared with the unmodified library. Also, results demonstrated that our solution is feasible and scalable, in terms of number of messages sent and message size. The major overhead is presented at bus connection step, and it can be reduced by using an enclave pool approach, as described by [25], and will be subject of future work.

Security assessment presented strong guarantees to data confidentiality and integrity, with Intel SGX providing a root of trust that allows to authenticate the key agreement procedure and the exchanged messages. Moreover, SGX enclaves provide an isolated environment that allow to handle the keys and perform the encryption and decryption procedures in a secure way.

Our solution is limited only to sending unicast messages, since the Intel SGX architecture does not provide mechanisms to perform group attestation, which makes it difficult to create a secure communication channel that enables multicast message sending. In the same way, broadcast messages are not addressed in our approach. These subjects will be addressed in future work.

Finally, our solution can be applied to any message bus and can be extended to other IPC mechanisms. Also, other security mechanisms can be used to provide trust execution, both hardware-based [28] or software-based [29]. Also, this approach can be used in distributed or cloud environments, by applying the trusted execution approach in network communication.

Acknowledgment

This research was developed in the context of the H2020 - MCTI/RNP *Secure Cloud* project. The authors also thank the UFPR and UTFPR Computer Science departments.

References

[1] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "User-level interprocess communication for shared memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 2, pp. 175–198, 1991.

[2] M. D. Cvetkovic and M. S. Jevtic, "Interprocess communication in real-time Linux," in *Proceedings of the 6th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Service*. Nis, Yugoslavia: IEEE, 2003, pp. 618–621.

[3] S. R. Shimko and J. Brindle, "Securing inter-process communications in SELinux," in *Proceedings of the 3rd Annual Security Enhanced Linux Symposium*, Baltimore, MD, USA, 2007.

[4] A. Venkataraman and K. K. Jagadeesha, "Evaluation of inter-process communication mechanisms," *Architecture*, vol. 86, p. 64, 2015.

[5] Z. Spasov and A. M. Bogdanova, "Inter-process communication, analysis, guidelines and its impact on computer security," in *Proceedings of the 7th International Conference for Informatics and Information Technology*. Bitola, Macedonia: Institute of Informatics, 2010.

[6] H. Pennington, A. Carlsson, A. Larsson, S. Herzberg, S. McVittie, and D. Zeuthen, *D-Bus specification*, freedesktop.org, 2020, rev. 0.36.

[7] Intel, *Intel Software Guard Extensions SDK for Linux OS Developer Reference*, Intel Corporation, 2016.

[8] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Tel-Aviv, Israel: ACM, 2013.

[9] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "SecureStreams: A reactive middleware framework for secure data stream processing," in *Proceedings of the 11th International Conference on Distributed and Event-based Systems*. Barcelona, Spain: ACM, 2017.

[10] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Tel-Aviv, Israel: ACM, 2013.

[11] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "SkyBridge: Fast and secure inter-process communication for microkernels," in *Proceedings of the 14th EuroSys Conference*. Dresden, Germany: ACM, 2019.

[12] D. Du, Z. Hua, Y. Xia, B. Zang, and H. Chen, "XPC: Architectural support for secure and efficient cross process call," in *Proceedings of the 46th International Symposium on Computer Architecture*. Phoenix, AZ, USA: ACM, 2019.

[13] J. Gu, X. Wu, W. Li, N. Liu, Z. Mi, Y. Xia, and H. Chen, "Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication," in *Proceedings of the USENIX Annual Technical Conference*. Boston, MA, USA: USENIX Association, 2020, pp. 401–417.

[14] S. L. Mirtaheri, E. M. Khaneghah, and M. Sharifi, "A case for kernel level implementation of inter process communication mechanisms," in *Proceedings of the 3rd International Conference on Information and Communication Technologies: From Theory to Applications*. Damascus, Syria: IEEE, 2008, pp. 1–7.

[15] F. S. Gharehchopogh, E. Amini, and I. Maleki, "A new approach for inter process communication with hybrid of message passing mechanism and event based software architecture," *Indian Journal of Science and Technology*, vol. 7, no. 6, pp. 839–847, 2014.

[16] X. Zhang, M. J. Covington, S. Chen, and R. Sandhu, "SecureBus: Towards application-transparent trusted computing with mandatory access control," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*. Singapore: ACM, 2007, pp. 117–126.

[17] D. Bernstein, "CurveCP: Usable security for the Internet," 2011. [Online]. Available: http://curvecp.org

[18] CurveZMQ, "ZeroMQ RFC 26/CURVEZMQ," Sep 2020. [Online]. Available: https://rfc.zeromq.org/spec:26/CURVEZMQ/

[19] N. C. Will, T. Heinrich, A. B. Viescinski, and C. A. Maziero, "A trusted message bus built on top of D-Bus," in *Proceedings of the XX Brazilian Symposium on Information and Computational Systems Security*. Petrópolis, RJ, Brazil: SBC, 2020.

[20] F. Akmel, E. Birhanu, B. Siraj, and S. Shifa, "A comparative analysis on software architecture styles," *International Journal in Foundations of Computer Science & Technology*, vol. 7, no. 5/6, pp. 11–22, 2017.

[21] N. C. Will and C. A. Maziero, "Using a shared SGX enclave in the UNIX PAM authentication service," in *Proceedings of the 14th Annual International Systems Conference*. Montreal, QC, Canadá: IEEE, 2020.

[22] T. Bui, S. P. Rao, M. Antikainen, V. M. Bojan, and T. Aura, "Man-in-the-Machine: Exploiting ill-secured communication inside the computer," in *Proceedings of the 27th USENIX Security Symposium*. Baltimore, MD, USA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bui

[23] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *Proceedings of the International Workshop on Public Key Cryptography*. New York, NY, USA: Springer, 2006, pp. 207–228.

[24] G. Paoloni, "How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures," *Intel Corporation*, 2010.

[25] D. Li, R. Lin, L. Tang, H. Liu, and Y. Tang, "SGXPool: Improving the performance of enclave creation in the cloud," *Transactions on Emerging Telecommunications Technologies*, p. e3735, 2019.

[26] A. T. Gjerdrum., R. Pettersen., H. D. Johansen., and D. Johansen., "Performance of trusted computing in cloud infrastructures with Intel SGX," in *Proceedings of the 7th International Conference on Cloud Computing and Services Science*. Porto, Portugal: SciTePress, 2017, pp. 696–703.

[27] C. Zhao, D. Saifuding, H. Tian, Y. Zhang, and C. Xing, "On the performance of Intel SGX," in *Proceedings of the 13th Web Information Systems and Applications Conference*. Wuhan, China: IEEE, 2016, pp. 184–187.

[28] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proceedings of the 25th USENIX Security Symposium*. Austin, TX, USA: USENIX Association, 2016, pp. 857–874.

[29] U. Lee and C. Park, "SofTEE: Software-based trusted execution environment for user applications," *IEEE Access*, vol. 8, pp. 121 874–121 888, 2020.