A Trusted Message Bus Built on Top of D-Bus

Newton C. Will, Tiago Heinrich, Amanda B. Viescinski, Carlos A. Maziero

Computer Science Department
Federal University of Paraná State (UFPR)
Curitiba – PR – Brazil

{ncwill,theinrich,abviescinski,maziero}@inf.ufpr.br

Abstract. *A wide range of applications use Inter-Process Communication (IPC) mechanisms to communicate between each other or between their components running in different processes. A well-known IPC mechanism in UNIX-like systems is D-Bus, which allows processes to communicate by receiving and routing messages. Despite being widely used, such system lacks mechanisms to provide end-to-end data confidentiality. In this paper we propose the use of Intel Software Guard Extensions (SGX) to provide a trusted communication channel between local applications over the D-Bus message bus system. We obtained stronger security guarantees in message confidentiality and integrity while keeping a small Trusted Computing Base (TCB) and compatibility with the reference D-Bus system.*

1. Introduction

A UNIX environment is composed of applications and services that frequently need to communicate with each other [Tanenbaum and Bos 2015, Lauer 2019]. This is accomplished using *Inter-Process Communication (IPC)* mechanisms, which allow the coordinated information exchange between such elements. Different IPC solutions exist, implementing different communication paradigms, with distinct concerns about performance, modularity, and scope.

The standard IPC mechanism for many UNIX-like desktop platforms is D-Bus [freedesktop.org 2018]. It is responsible to provide basic functionalities such as data exchange between processes and also more complex operations. Based on a client/server architecture, its offers two types of services: *bidirectional*, where a resource is requested through an exchange of messages, and *unidirectional*, where the information is broadcast to interested clients.

D-Bus messages are fully typed, the communication is based on a single message bus shared in the system, allowing the integration between events [Love 2005]. Shared bus provides a simple mechanism for exchanging messages, but, on the other hand, makes the solution susceptible to *denial-of-service (DoS)* attacks and may expose sensitive application data to malicious entities.

Confidentiality issues in the use of IPC mechanisms are described by [Bui et al. 2018], and can be addressed by using cryptographic protocols. These protocols can be implemented by the final applications or by the IPC itself. To reinforce the security constraints, a *Trusted Execution Environment (TEE)* can be used, in order to provide authentication and a root of trust to applications. Similar security concerns are explored in distributed IPC mechanisms [Pires et al. 2016, Havet et al. 2017].

This paper presents a framework that applies the Intel *Software Guard Extensions (SGX)* framework to protect the information exchange on D-Bus. The developed system allows applications to exchange messages with strong integrity and confidentiality guarantees, using security mechanisms provided by the hardware. Our prototype was evaluated and compared to the standard D-Bus implementations.

The text is organized as follows: Section 2 presents the background for the study; Section 3 presents previous research related to D-Bus and secure communication between processes; Section 4 describes the solution proposed by this paper; implementation of a proof of concept, as well performance evaluation and security assessment of the solution are presented in Section 5; finally, Section 6 concludes the paper.

2. Background

This Section presents a brief description of important concepts used in this work, such as D-Bus, D-Bus Broker, and the Intel *Software Guard Extensions (SGX)*.

2.1. D-Bus

D-Bus is a message bus system used as an IPC mechanism between multiple application in the same machine. Using D-Bus, applications can send messages to particular services and broadcast messages to all interested services; it is also used for communication between system programs and user sessions [Pennington et al. 2020]. D-Bus is the main IPC mechanism used in current Linux desktops.

D-Bus consists of a library (*libdbus*) that allows the connection and message exchange between applications; a daemon that allows the connection of several applications and performs the message routing; and high-level libraries that enable applications to communicate with *libdbus*. The D-Bus daemon provides two buses, a *system bus* and a *session bus*, in which the process can connect and use the message routing service, as shown in Fig. 1.

The system bus is used for applications to interact with system components without dealing with low-level system details. It can provide important information to applications, such as adding new hardware, low battery alert, or network status. The session bus allows the user applications to share data and event notifications between them in a single user session. Any user process can connect to the system bus and to its current session bus, but not to another users' session buses.

When each application connects to the bus, the daemon assigns a unique identifier to the process, ensuring that identifier is never reused during the lifetime of the bus daemon. In order to be located by others, each application must register a known name on the bus, the *service name*, and the daemon will perform the translation of the service name into the identifier.

D-Bus messages have two sections. The first section, *header*, has a set of information used in the message routing, such as sender, destination, message type, and also the type signature for the data. The second section, *body*, has the message data in binary format. There are also two main types of messages: *method* and *signal*. Methods are operations that can be invoked in the destination and can contain optional arguments and return values. Signals are used by the application to notify others of the occurrence of an

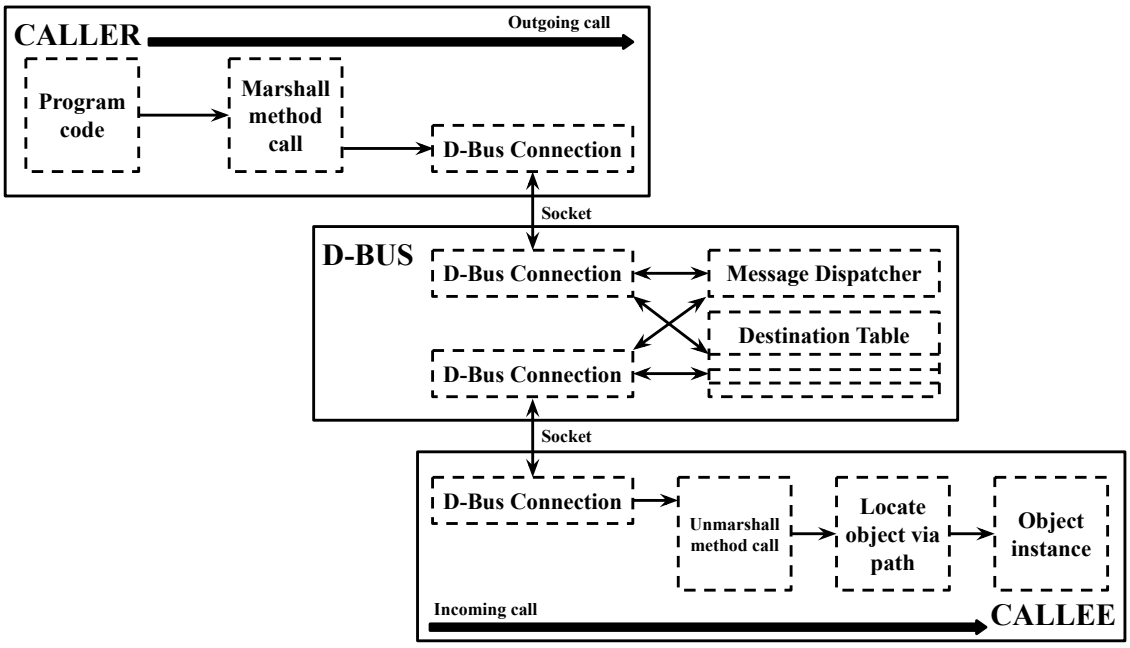

Figure 1. D-Bus overview [Marhefka and Muller 2014].

event and will be received by all processes connected to the bus. Signals can also contain arguments, and they do not expect any reply.

2.2. D-Bus Broker

An alternative to the default implementation of D-Bus is *D-Bus Broker* [bus1 2018], which is a drop-in replacement for the D-Bus reference implementation and keeps compatibility to it. D-Bus Broker uses a set of modern features provided by recent Linux kernel releases and an improved implementation to provide higher performance, reliability, scalability, and security.

The standard D-Bus daemon performs connection accounting on a process basis. Clients can circumvent this accounting by creating multiple processes, in order to get new connection quotas, opening the way to DoS attacks. The D-Bus Broker implementation uses a user-based accounting, preventing this attack.

All internal errors are considered fatal by D-Bus Broker; when an error occurs, all peers are disconnected and the broker is shut down, with no tries to work around the situation. This behavior avoids to put peers into unexpected situations or to silently drop messages. Also, when a peer commits a protocol violation, it receives an error reply and is then disconnected.

In order to avoid deadlocks, D-Bus Broker do not use IPC methods in its implementation. The operation of message transactions is a self-contained procedure, without any external hooks nor callbacks. Also, D-Bus Broker does not employ any global data-structures non-required by D-Bus specification, with message transactions only being affected by the data provided by the involved peers. This makes all lookups in D-Bus Broker to take $O(log(n))$ time, unlike D-Bus daemon, which takes $O(n)$ times in several situations.

Finally, in addition to the use of D-Bus Broker as a drop-in replacement for the reference implementation, it is also possible to integrate the message broker as an isolated process, without compatibility restrictions and without any side-effects or file-system access, behaving like a library and being perfectly suited for private buses.

2.3. Security Issues in D-Bus

Applications using D-Bus are subject to some problems, such as memory leakage and data loss, as shown by [Marhefka and Muller 2014], because of the way the message exchanges take place, since the application must validate the received data as it does with unreliable inputs. Problems of this nature affect applications performance and threaten reliability, as they are vulnerable to attacks such as DoS [Whittaker 2002].

An improvement is *kdbus*, an in-kernel implementation of the transport layer, taking advantage of Linux kernel features to overcome D-Bus limitations imposed by the user-space implementation [freedesktop.org 2015]. It brings more security to the system, being available at boot time, providing reliable metadata transmission, a simplified policy mechanism, and allowing zero-copy messages between process message exchange [Atlidakis et al. 2016].

Besides that, several applications may still be vulnerable to attacks that compromise data confidentiality when using IPC mechanisms. An attacker may have access to the system and listen the message bus in order to retrieve messages exchanged between processes. This type of attack is defined as *Man-in-the-Machine*, and can reveal sensitive users' information, such as passwords and authentication tokens [Bui et al. 2018].

The security of D-Bus can be improved by requiring a bus authentication, ensuring that applications can only connect to it when presenting the appropriate credentials. D-Bus authentication is based on the *Simple Authentication and Security Layer (SASL)* standard [Melnikov and Zeilenga 2006], but it is not mandatory. It is also possible to apply a security policy, which allows the filtering of exchanged data by characteristics such as source, recipient, and content. SELinux can also be enabled in D-Bus in two ways: messages that are routed have their permission checked; and when a message asks to own a name, the context is verified. Despite this, end-to-end data protection is left to applications [freedesktop.org 2020].

2.4. Intel Software Guard Extensions

Intel SGX is an extension of the 6th+ Intel CPU instruction set. The new instructions protect sensitive information in an application that could be modified or accessed by software running at a higher privilege level. With such instructions, developers are allowed to define private regions of memory, that are known as *enclaves*, blocking any access from outside of enclaves in this data. The main goal of the SGX architecture is to reduce the *Trusted Computing Base (TCB)* to a piece of hardware and software, as shown in Fig. 2.

Each enclave has an author self-signed certificate containing information that allows SGX to detect when any part of the enclave has been tampered with, allowing an enclave to prove that it has been correctly loaded to the memory and is trustworthy. The enclave creation consists of several steps: initialization of the control structure of the enclave; page memory allocation and loading the enclave content to these pages; measurement of the enclave content; and creating an enclave identifier. The enclave is

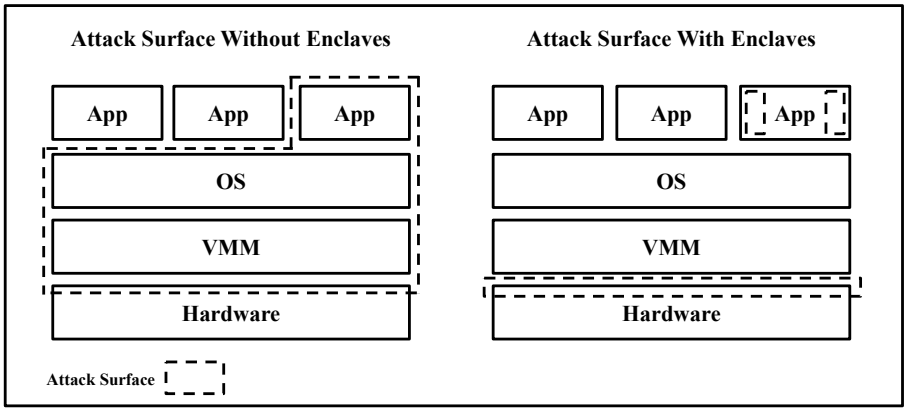

Figure 2. Attack surface of a security-sensitive application without and with SGX enclaves [Sobchuk et al. 2018].

linked to the application that created it, which is open to any inspection and analysis [McKeen et al. 2013, Costan and Devadas 2016].

Unauthorized access to data or code inside an enclave are blocked, generating access faults. The data in the enclave memory is encrypted, with the encryption key being stored in the CPU, without access by external entities [Costan and Devadas 2016, Intel 2016]. While the data is being transferred between the registers, unauthorized access is avoided using the internal access control mechanisms of the processor itself. Malware and even system code with higher privilege can't change the data inside the enclave, ensuring its confidentiality and integrity [McKeen et al. 2013, Jain et al. 2016].

Enclaves are also able to share data each other by using an attestation mechanism, which allows an enclave to prove that it is legitimate, has not been tampered with and was loaded correctly, allowing the creation of a secure channel for communication. Local attestation is used when both enclaves are running in the same platform, defining a symmetric key by a Diffie-Hellman key agreement procedure, authenticated by the hardware. This procedure ensures that both enclaves were loaded correctly and that all static measurements are valid. Remote attestation is also provided [Anati et al. 2013].

3. Related Work

There is a lack of proposals to improve the security of the D-Bus infrastructure. The current section discusses some papers found in the literature that cover this subject.

The structure used in D-Bus allows users to call methods with a set of arguments. [Marhefka and Muller 2014] developed a tool name *Dfuzzer* to perform fuzz testing of D-Bus services. As a client, Dfuzzer calls remote methods through the D-Bus interface, with the types and values of the arguments being automatically discovered. The monitor is responsible for watching the calls and detecting odd behavior.

There are some works indented to complement D-Bus or to replace it by more robust and efficient solutions. One is *Bus1 Kernel Message Bus*, an in-kernel IPC subsystem focused on high performance, compatibility, and reliability. It pushes into the kernel some features of D-Bus to improve their performance [bus1 2016]. *D-Bus Broker*, presented in Section 2.2, also replaces D-Bus by a more efficient solution. None of them deals with

message confidentiality or integrity issues [bus1 2018].

Other IPC frameworks are frequently used to develop applications. A popular one is *ZeroMQ* [ZeroMQ 2020], a lightweight messaging bus used in many organizations, like Spotify, Microsoft, and Samsung. It allows users to connect their code using any language or platform, with a focus on stability and reliability. ZeroMQ uses the sockets API, allowing its use in distributed applications.

The paper [Pires et al. 2016] uses Intel SGX to create secure communication channels, and ZeroMQ is used on them to implement a secure content-based routing mechanism between enclaves. They achieved acceptable performance, as long as the memory used by each enclave is kept within the size of its private memory. Similarly, [Havet et al. 2017] applies ZeroMQ and SGX in a middleware framework for data flow processing tasks. Its goal is to provide end-to-end security guarantees in industrial-level data processing, such as large-scale clusters or cloud infrastructures. The proposal has limited scalability related to the availability of physical cores.

We did not find any research work intended to improve the security aspects of communication using D-Bus, concerning the confidentiality and integrity of messages exchanged by applications. This motivated us to investigate how the Intel SGX framework could be used in such context.

4. A Trusted Message Bus

D-Bus is a well-known Linux *Inter-Process Communication (IPC)* mechanism, enabling applications to communicate each other and with system modules. One of the characteristics of message buses (session bus and system bus) is that both are public, allowing applications to connect to them and to listen the messages. We can use the *dbus-monitor* tool to listen all messages and data put on each bus.

Some applications may share sensitive data with other application or with a system service, such as in authentication systems [Will and Maziero 2020] and password managers [Bui et al. 2018], which require data confidentiality. Confidentiality can be achieved by using asymmetric or symmetric encryption, but both solutions lack authentication when there is no root of trust, enabling the occurrence of man-in-the-middle attacks.

In this work, we propose the use of Intel SGX as a root of trust to establish secure communications between applications using D-Bus. Our proposal uses the local attestation mechanism to establish a communication session between the applications, with the key agreement procedure authenticated by the hardware, avoiding man-in-the-middle attacks, even their active forms.

Our implementation includes the *Trusted D-Bus Library* (*tdbus*)[1], built on top of *libdbus*, to perform the attestation and data ciphering procedures. Fig. 3 shows the architecture overview of our solution, in which Fig. 3(a) presents a simple usage of *libdbus* reference implementation and Fig. 3(b) presents the organization and communication flow among *tdbus*, *libdbus*, and the enclave; the encrypted data flow is represented by a lock.

[1]The source code of our prototype is available at `https://github.com/newtoncw/tdbus`.

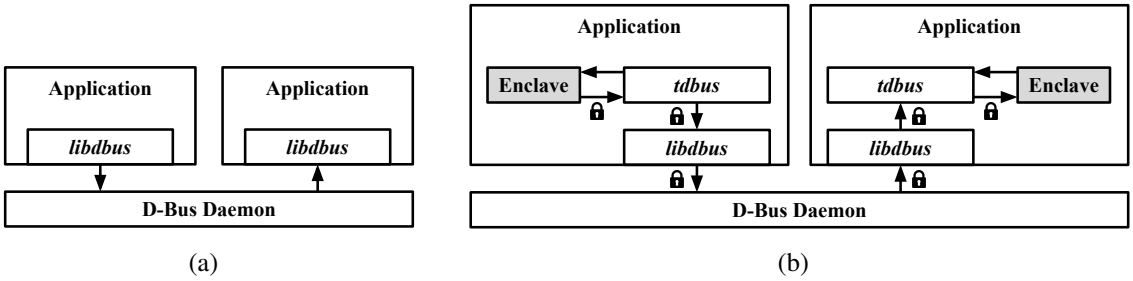

Figure 3. (a) Usage of *libdbus* reference implementation by the application; (b) Communication flow among *tdbus*, *libdbus*, and the enclave.

4.1. Service Registration and Secure Session Setup

As described in Section 2.1, connection establishment in D-Bus and D-Bus Broker involves the client request to use a name (*service name*) that identifies it on the message bus, with D-Bus daemon and D-Bus Broker using this name to route messages to this client.

Our solution includes the enclave creation at this step, which will carry out the attestation and data encryption/decryption. In order to provide a secure end-to-end communication channel, we add an step to perform an authenticated Diffie-Hellman key agreement between the two applications that will exchange data, by using the local attestation procedure provided by Intel SGX, thus creating a secure communication session between the applications. Each *service name* registered by the application creates an enclave, and each enclave is able to handle multiple communication sessions.

To establish the communication session, both application enclaves attest each other using the mechanisms provided by Intel SGX, ensuring that both are running in the same platform. Attestation also provides a certificate for the key agreement procedure, allowing to validate data received in this process. Key agreement is performed by a three-step Diffie-Hellman procedure, with one client sending its identification to the other, which will use this identification to generate a signed report that will be send back to the first client. It will then validate the received report and send a signed response to the other client, to be validated there. All these steps are done through and open channel and at the end both clients will have a 128-bit session key that will be used to encrypt all messages exchanged between them.

4.2. Secure Communication

Each communication session ensures an end-to-end secure channel, with a specific key used to cipher data. The session key defined will be used to cipher all data exchanged between the two applications until the session is closed. This session key is handled only by the enclave and never leaves its boundaries, as well as the data encryption/decryption procedures, ensuring their integrity. Attestation and data encryption are transparent to the programmer, being carried out by the *tdbus* library, which contains all the code to implement such tasks.

Data cipher includes the entire message body section and the type signature for the data, preventing the attacker from listening what data and data types are being exchanged between applications. The remaining information in message header section cannot be

encrypted, since they are necessary for the message routing that is carried out by the D-Bus daemon. This approach makes the message body opaque to D-Bus daemon and does not require any changes in D-Bus itself.

The data flow in the trusted message bus is presented in Fig. 4, including the data encryption procedures, performed inside enclaves (shaded boxes). The data structures to be sent are converted to a byte array and then encrypted using the session key. This byte array is then sent to D-Bus as an opaque message. We opted to add our own marshaller before the encryption, to keep the D-Bus libraries and runtime intact.

Then, the message is marshalled by D-Bus with the encrypted data array as a payload, routed by D-Bus daemon, and then received by the second application, which will unmarshall the message, decrypt the data inside the enclave, and rebuild the message structure as defined by the sender, allowing the application to use the data.

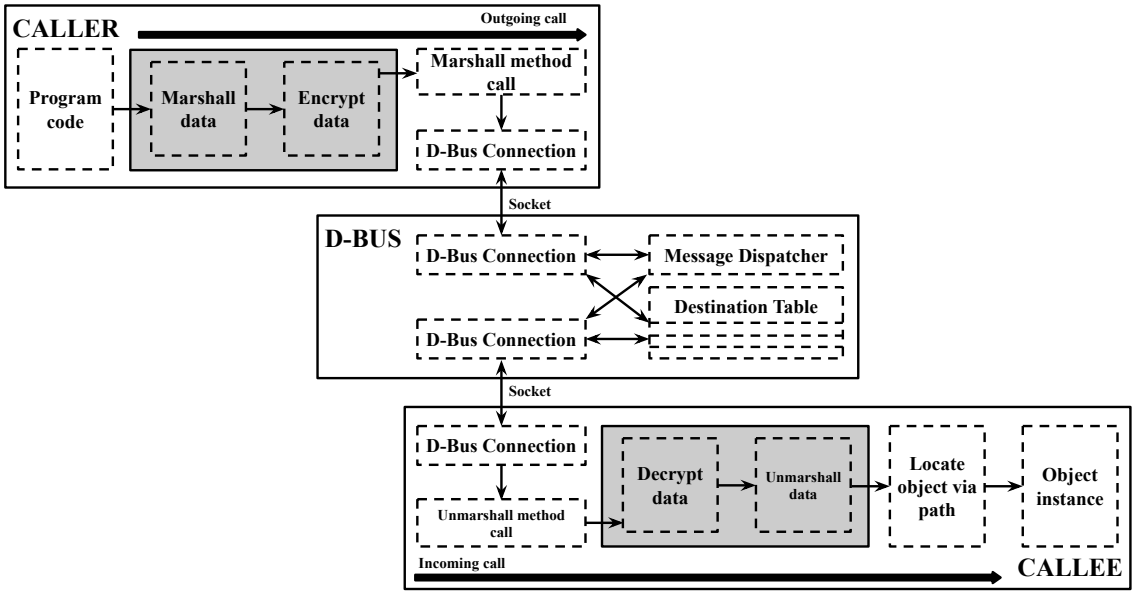

Figure 4. Trusted message bus overview. Shaded boxes indicate enclaves.

5. Evaluation

In order to evaluate the feasibility of our proposal, we implemented a prototype of the Trusted D-Bus Library using the C language. In this Section we discusses the performance trade offs and the security guarantees provided by the solution.

5.1. Performance Evaluation

We evaluated the performance of our solution in three different scenarios, applying it on both the standard D-Bus daemon and the D-Bus Broker solutions. Tests were run on a Dell Inspiron 7460 laptop, 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, running Ubuntu 18.04 LTS, kernel 4.15.0-109-generic. Intel TurboBoost, SpeedStep, and HyperThread extensions were disabled, to provide stable results. We used the D-Bus daemon 1.12.2; D-Bus Broker v23; Intel SGX SDK 2.9.101.2 and set the enclave stack size at 8 KB and the enclave heap size at 64 KB. All benchmarks were taken by using RDTSCP instruction

[Paoloni 2010]. Experiments were run 100,000 times each. The width of the confidence interval at 95% is 0.96% of the average.

The first evaluated scenario involves the connection establishment, in which we have major changes and we expect the biggest overhead. We started two applications, both register themselves as services in D-Bus and the first one starts a secure connection to the other. We measured the time needed for the service name registration and secure session establishment. Due to the enclave creation and attestation, the secure session establishment imposes a high overhead in our solution, around 7.7× using the D-Bus daemon and 9.3× with the D-Bus Broker, as shown by Fig. 5.

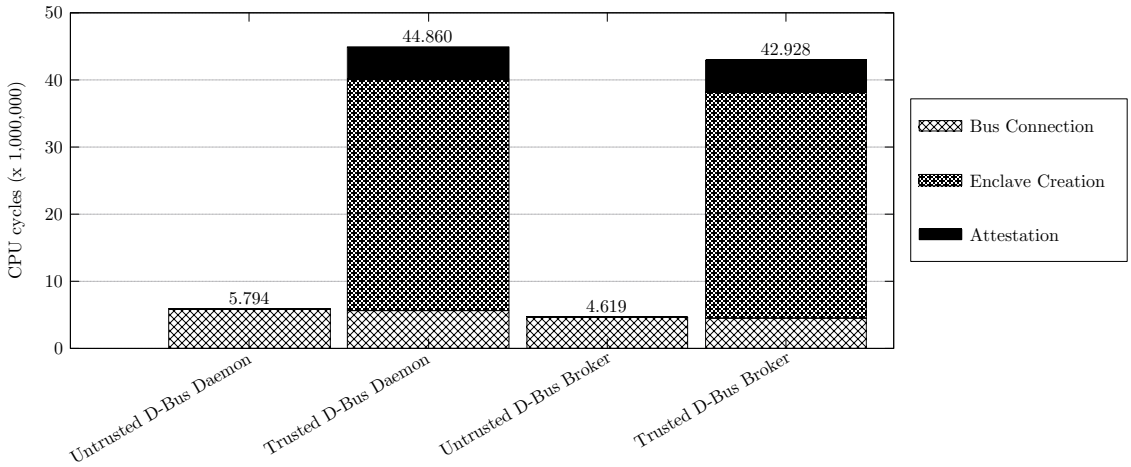

Figure 5. Connection/session establishment latency.

The secure session establishment includes the enclave creation, which is done when the application requests the *service name* to the bus, and is responsible for 77% of the time spent in this step. In consequence, the overhead for an application to establish new communication sessions after being connected to the bus will be considerably lower, as each enclave can handle multiple communication sessions. The time spent in the attestation process is around 4.8 million CPU cycles.

In order to evaluate the overhead of our marshalling and encryption/decryption process, we performed tests using four primitive data types (`uint8`, `uint16`, `uint32`, and `uint64`), with the results being presented by Fig. 6. In this scenario, our proposal imposes an overhead around 160% when compared to vanilla D-Bus daemon and D-Bus Broker.

In order to evaluate the influence of message size on the communication costs, we ran another test, in which a byte array of different sizes is sent. The results obtained are shown in Fig. 7. We can observe that D-Bus daemon and D-Bus Broker increase the processing time with larger arrays (above 4 KB to D-Bus daemon, and above 256 B to D-Bus Broker). We can see that our proposal adds an average overhead of 2.5×, to both B-Dus daemon and D-Bus Broker, similar to the results obtained in sending primitive types. This overhead is reduced in D-Bus Broker when the array size is equal or greater than 1 KB. Our experiments shows an overhead of 1.25× in D-Bus Broker with a 256 KB array.

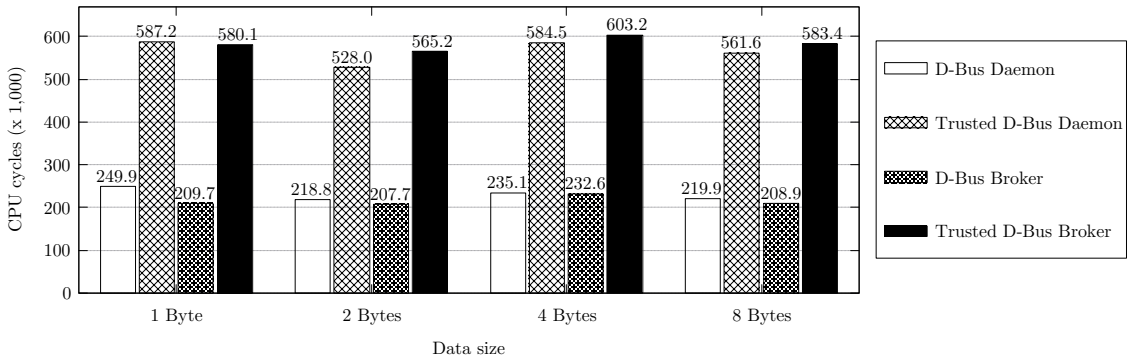

Figure 6. Latency in sending messages containing primitive data type.

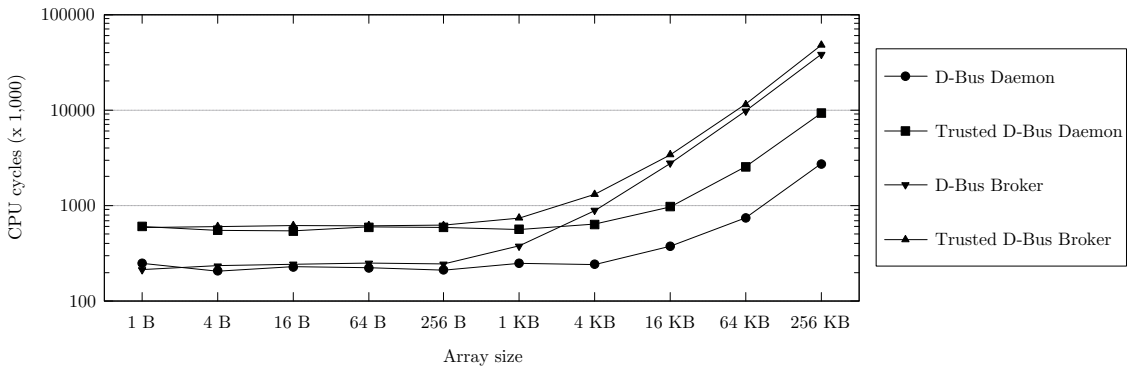

Figure 7. Latency in sending messages containing a data array.

Finally, we evaluated the cost of encryption and decryption procedures when sending messages, without considering marshalling (Fig. 8). The time spent in encryption/decryption steps increases with the array size, and it is responsible for about 56% of the total overhead imposed by out solution. Thus, we can achieve a better performance by improving marshalling procedure and avoiding memory copying operations.

5.2. Security Assessment

To build a system intended to be secure, its *Trusted Computing Base (TCB)* should be kept as small as possible, in order to reduce the chances of success in an attack. In Intel SGX technology, the TCB is composed of the CPU and its internal elements, such as hardware logic, microcode, registers, and cache memory.

Our solution keeps a small TCB with a single library (*tdbus*) built with less than 1,000 lines of code (of which only 230 lines of code in the enclave), which runs over the standard *libdbus* library. We split the *tdbus* library into two components: trusted and untrusted. The trusted component is responsible for the attestation procedure, encryption and decryption process, and handling session keys. The untrusted component contains the public interface for the applications and encapsulates the message data to be encrypted.

The *tdbus* trusted component runs inside an SGX enclave, which provides an encrypted memory area and ensures protection against external attacks, even if they come from components with high execution privilege, such as BIOS or hypervisor. Session

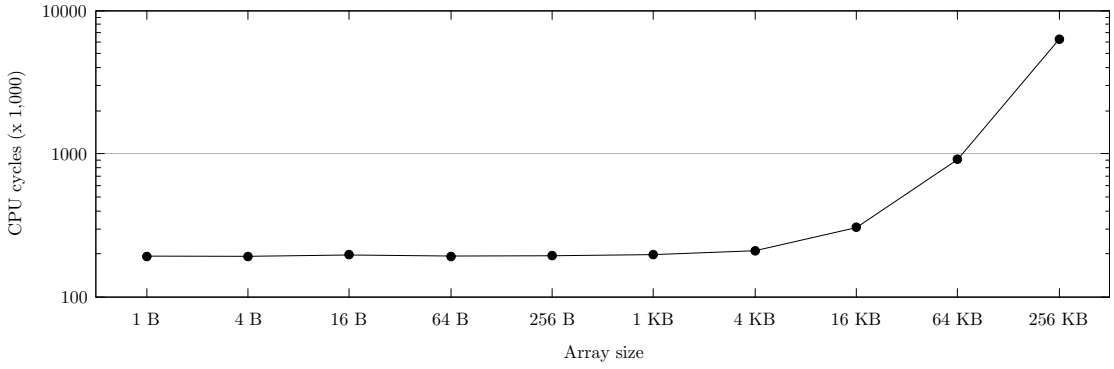

Figure 8. Latency in encryption/decryption procedures.

keys are kept inside the enclave and never go outside its boundaries, which ensures that they cannot be manipulated by any untrusted entity.

The attestation procedure ensures an authenticated key agreement when establishing the communication session, and this session provides an end-to-end secure communication channel. The use of an authenticated key agreement prevents passive and active man-in-the-middle and related attacks, reducing the range of possible attacks on the system aiming at listening or replacing data travelling on the bus.

Finally, encryption and decryption procedures are performed using an authenticated 128 bit AES-GCM algorithm, which ensures that any change in the message payload will be detected by the message receiver, causing the message to be dropped and the sender to be notified. Also, even if the attacker listens to the bus and collects the messages, she will spend a considerable time to crack the encryption, even in a high performance system. In addition, a session key must be used only for short periods of time.

6. Conclusion

This work proposed the use of the Intel *Software Guard Extensions (SGX)* technology to create a trusted version of the D-Bus *Inter-Process Communication (IPC)* mechanism, by adding a transparent end-to-end attestation procedure over the reference *libdbus* implementation. Our proposal creates a secure communication channel between processes, which allows sending sensitive information to each other while keeping the content opaque to the underlying message bus.

Performance evaluation experiments show a bigger overhead in the connection establishment, due the enclave creation cost, which is reduced when new trusted communication sessions are established using the same service name. The security assessment demonstrated that we can provide strong secure guarantees while keeping a very small trusted computing base, thus reducing the attack surface. Also, our solution is fully compatible with the standard D-Bus daemon, D-Bus Broker, and their libraries, and do not require any changes in these components.

An important limitation of our solution is related to SGX private memory size, limited to 128 MB in the current SGX version. This memory region includes metadata that is used to perform the enclave access control routines. Only about 90 MB of this protected memory are actually available for storing enclave data and code

[Shaon et al. 2017, Fuhry et al. 2017], which imposes a limitation on the amount of user data that can be handled by the enclave at same time.

As for future work, we intend to implement a new scheme to encrypt/decrypt data to overcome the limitation cited above, by splitting large amounts of data in smaller ones. Also, in order to achieve better performance, we will move our implementation to *libdbus*, to perform the data encryption/decryption after the native marshalling/unmarshalling procedure, ensuring the full integration with D-Bus.

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

Anati, I., Gueron, S., Johnson, S. P., and Scarlata, V. R. (2013). 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.

Atlidakis, V., Andrus, J., Geambasu, R., Mitropoulos, D., and Nieh, J. (2016). POSIX has become outdated. *;login:*, 41(3).

Bui, T., Rao, S. P., Antikainen, M., Bojan, V. M., and Aura, T. (2018). Man-in-the-machine: Exploiting ill-secured communication inside the computer. In *Proceedings of the 27th USENIX Security Symposium*, Baltimore, MD, USA. USENIX Association.

bus1 (2016). bus1 – Kernel Message Bus. `https://bus1.org/bus1.html`.

bus1 (2018). D-Bus Broker. `https://github.com/bus1/dbus-broker/wiki`.

Costan, V. and Devadas, S. (2016). Intel SGX explained. Cryptology ePrint Archive, Report 2016/086. `https://eprint.iacr.org/2016/086.pdf`.

freedesktop.org (2015). kdbus. `https://www.freedesktop.org/wiki/Software/systemd/kdbus/`.

freedesktop.org (2018). D-Bus. `https://www.freedesktop.org/wiki/Software/dbus/`.

freedesktop.org (2020). dbus-daemon. `https://dbus.freedesktop.org/doc/dbus-daemon.1.html`.

Fuhry, B., Bahmani, R., Brasser, F., Hahn, F., Kerschbaum, F., and Sadeghi, A.-R. (2017). HardIDX: Practical and secure index with SGX. In *Proceedings of the XXXI Data and Applications Security and Privacy*, Philadelphia, PA, USA. Springer.

Havet, A., Pires, R., Felber, P., Pasin, M., Rouvoy, R., and Schiavoni, V. (2017). 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.

Intel (2016). *Intel Software Guard Extensions SDK for Linux OS Developer Reference*. Intel Corporation. `https://01.org/sites/default/files/`

documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf.

Jain, P., Desai, S., Kim, S., Shih, M.-W., Lee, J., Choi, C., Shin, Y., Kim, T., Kang, B. B., and Han, D. (2016). OpenSGX: An open platform for SGX research. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA. Internet Society.

Lauer, M. (2019). *D-Bus*, pages 171–200. Apress, Berkeley, CA, USA.

Love, R. (2005). Get on the D-BUS. *Linux Journal*, 2005(130):3.

Marhefka, M. and Muller, P. (2014). Dfuzzer: A D-Bus service fuzzing tool. In *Proceedings of the 7th International Conference on Software Testing, Verification and Validation Workshops*, Cleveland, OH, USA. IEEE.

McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. (2013). 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.

Melnikov, A. and Zeilenga, K. (2006). Simple Authentication and Security Layer (SASL). RFC 4422, RFC Editor. https://tools.ietf.org/html/rfc4422.

Paoloni, G. (2010). How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation*. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

Pennington, H., Carlsson, A., Larsson, A., Herzberg, S., McVittie, S., and Zeuthen, D. (2020). *D-Bus specification*. freedesktop.org. rev. 0.36. https://dbus.freedesktop.org/doc/dbus-specification.html.

Pires, R., Pasin, M., Felber, P., and Fetzer, C. (2016). Secure content-based routing using Intel Software Guard Extensions. In *Proceedings of the 17th International Middleware Conference*, Trento, Italy. ACM.

Shaon, F., Kantarcioglu, M., Lin, Z., and Khan, L. (2017). SGX-BigMatrix: A practical encrypted data analytic framework with trusted processors. In *Proceedings of the Conference on Computer and Communications Security*, Dallas, TX, USA. ACM.

Sobchuk, J., O'Melia, S., Utin, D., and Khazan, R. (2018). Leveraging Intel SGX technology to protect security-sensitive applications. In *Proceedings of the 17th International Symposium on Network Computing and Applications*, Cambridge, MA, USA. IEEE.

Tanenbaum, A. S. and Bos, H. (2015). *Modern Operating Systems*. Pearson, Boston, MA, USA, 4th edition.

Whittaker, J. A. (2002). *How to break software*. Addison-Wesley, Boston, MA, USA.

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

ZeroMQ (2020). An open-source universal messaging library. https://zeromq.org.