Secure Cloud Micro Services Using Intel SGX

Stefan Brenner, Tobias Hundt, Giovanni Mazzeo, Rüdiger Kapitza

Secure Cloud Micro Services using Intel SGX

Stefan Brenner[1], Tobias Hundt[1], Giovanni Mazzeo[2], and Rüdiger Kapitza[1]

[1] TU Braunschweig, Germany, {brenner,hundt,rrkapitz}@ibr.cs.tu-bs.de
[2] University of Naples "Parthenope", Italy, giovanni.mazzeo@uniparthenope.it

Abstract. The micro service paradigm targets the implementation of large and scalable systems while enabling fine-grained service-level maintainability. Due to their scalability, such architectures are frequently used in cloud environments, which are often subject to privacy and trust issues hindering the deployment of services dealing with sensitive data.
In this paper we investigate the integration of trusted execution based on Intel Software Guard Extensions (SGX) into micro service applications. We present our *Vert.x Vault*, that supports SGX-based trusted execution in Eclipse Vert.x, a renowned tool-kit for writing reactive micro service applications. With our approach, secure micro services can run alongside regular ones, inter-connected via the Vert.x event bus to build large Vert.x applications that can contain multiple trusted components.
Maintaining a full-fledged Java Virtual Machine (JVM) inside an SGX enclave is impractical due to its complexity, less secure because of a large Trusted Code Base (TCB), and would suffer from performance penalties due to a high memory footprint. However, as Vert.x is written in Java, for a lean TCB this requires integration of native enclave C/C++ code into Vert.x, for which we propose the usage of Java Native Interface (JNI).
Our *Vert.x Vault* provides the benefits of micro service architectures together with trusted execution to support privacy and data confidentiality for sensitive applications in the cloud at scale. In our evaluation we show the feasibility of our approach, buying a significantly increased level of security for a low performance overhead of only $\approx 8.7\%$.

Keywords: Vert.x, SGX, Cloud Security, Micro services

1 Introduction

Micro services are popular as they offer a new paradigm and many benefits such as flexibility, scalability, ease of development and manageability of applications [3]. Due to their scaling nature and flexibility, micro service architectures are mostly used in data centre and cloud scenarios where scale out capabilities are required to handle high load. However, trust issues still hinder the widespread adoption of cloud services and the deployment of sensitive applications processing sensitive data in the cloud [11,13].

With Software Guard Extensions (SGX) [4,12], Intel recently released a new technology for protecting applications from many—even physical—attacks such as the cold boot attack. SGX is an instruction set extension, released in

the Skylake processor family, that allows the creation of Trusted Execution Environment (TEEs) inside the address space of an application. SGX TEEs are called *enclaves* and provide strong protection of code and data inside through encryption and integrity checks of their memory range directly by the CPU. This allows a strong adversary model and to limit the trusted computing base to the enclave code and the CPU package only, which is especially useful for sensitive data processing in the cloud. On currently available hardware, SGX is limited to a maximum of 128MB of memory to be used for enclaves. This limitation requires to keep the memory footprint of enclaves as narrow as possible in order to prevent significant performance implications. Another important aspect with regards to security of TEEs is to keep their Trusted Code Base (TCB) as small as possible. This is due to the fact, that larger amounts of code usually lead to more exploitable security vulnerabilities [15].

Amongst others like *Spring Boot*[1], *Go Micro*[2], an excellent example for a renowned protagonist in the context of micro service development is Eclipse Vert.x [2]. The Vert.x tool-kit introduces the notion of *verticles* and supports the development of micro service applications. Furthermore, it provides a distributed event bus for its verticles to communicate with each other in a reactive fashion.

In this paper, we investigate the protection of data confidentiality in micro service applications by exploitation of the SGX technology, and chose the Vert.x tool-kit as an example micro service environment. Our contribution comprises the design of our *Vert.x Vault*, that allows the integration of slim SGX TEEs into the JVM-based Vert.x tool-kit, and demonstrates the feasibility of our approach, as well as an evaluation of its induced performance overhead.

A technical challenge to be solved in this work was the way of integration of native C/C++-based TEEs into the Java/Java Virtual Machine (JVM)-based environment of the Vert.x tool-kit. As we argued in earlier work [8], porting a full-fledged JVM into a TEE is not only a complex endeavour, but also violates common security principles such as the size and complexity of the TCB as well as the memory footprint of the TEEs which is highly performance-critical. Moreover, in this work we elaborate why the usage of SGX-based TEEs is a good fit for micro service applications.

In Section 2 we describe SGX and Vert.x as the cornerstone of our system. Next, in Section 3 we show the design of our *Vert.x Vault* followed by implementation details in Section 4. Afterwards, in Section 5, we present the benefits of our *Vert.x Vault* adopted in a critical infrastructure use case scenario. Finally, we measure the performance of our approach in Section 6 and conclude in Section 8.

2 Background

In this section we describe the architecture, purpose, and main concepts of the Vert.x tool-kit. In addition, we give a short introduction to the SGX technology and describe the enclave life cycle and programming model.

[1] `https://projects.spring.io/spring-boot/`
[2] `https://github.com/micro/go-micro`

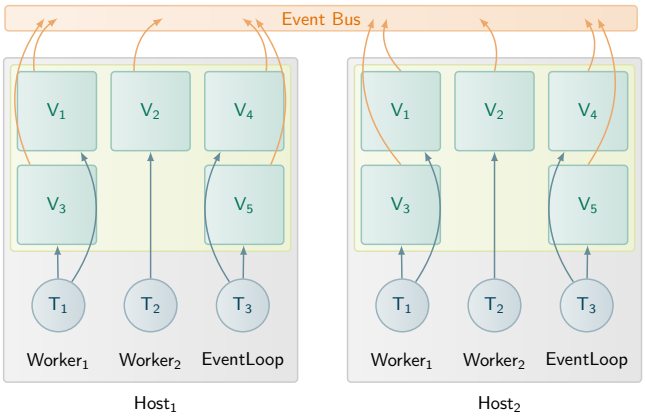

Fig. 1: Vert.x architecture and thread model (Verticles V_{1-5}, Threads T_{1-3}).

2.1 Eclipse Vert.x

The rationale behind the concept of a micro service is to do one small thing and reduce complexity. In a micro service architecture, multiple of these services interact with each other and each of them has its very limited purpose. This allows the flexibility of development of such architectures, and also large development teams to collaborate simultaneously. Real applications of Vert.x can be for example REST services, or real time web applications[3].

According to its authors, "Vert.x is a tool-kit for building reactive applications on the JVM" [2]. In the context of Vert.x, micro services are called *verticles*, and are supposed to comprise a scarce, well-defined part of application logic. As Vert.x is a polyglot tool-kit for JVM-based languages, verticles can be implemented in various programming languages and interact across programming languages. Verticles communicate via the *event bus* that connects verticles even across machine boundaries were they can subscribe an "address" in order to receive callbacks once a message arrives for this address.

All verticles are scheduled via the *event loop* thread of Vert.x (one per physical CPU core), which delivers events to verticles. Supporting the idea of reactive applications, a verticle should never block this thread but implement blocking operations such as I/O operations in an asynchronous fashion. Long running tasks can be done in a *worker verticle* on a separate thread pool. Figure 1 illustrates the overall Vert.x system architecture and thread model.

Currently, there is no support for trusted execution in Vert.x, however, the Vert.x engineers recently released a secure event bus implementation that provides protection of exchanged event bus messages using Transport Layer Security (TLS).

[3] `http://vertx.io/whos_using/`

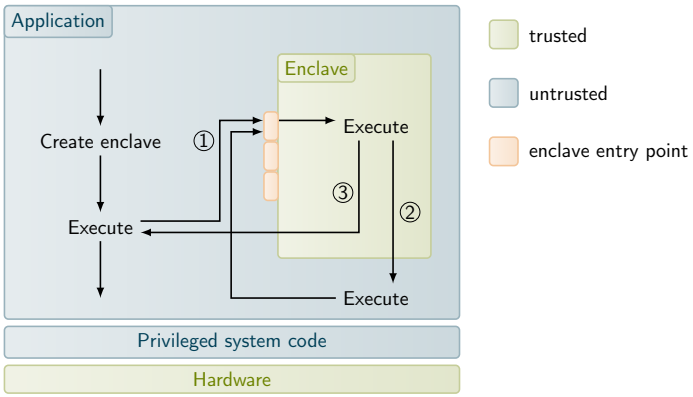

Fig. 2: Interaction between untrusted application and SGX enclave.

2.2 SGX

SGX [4,12] is a new instruction set extension by Intel, which has been released in the Skylake processor generation. It allows the creation of TEEs—called *enclaves*—inside the address space of user space applications. Enclaves are protected by the CPU package itself from a number of critical security threats: enclave memory is transparently encrypted and integrity-checked by the CPU, limiting trust to only the CPU package itself.

Entering and exiting an enclave is only possible via a defined enclave interface, that describes entry points and the allowed number of concurrent threads inside an enclave. Calls to enclave functionality are called enclave calls (ecalls), while calls from an enclave to outside code are called outside calls (ocalls). The enclave interface is described in a domain-specific language during development, which is also used to generate untrusted and trusted ecall and ocall stubs. Figure 2 illustrates the control flow of an ecall ①, an ocall ② and returning from an ecall to the untrusted code ③.

Enclave memory is backed by a range of normal DRAM—called Enclave Page Cache (EPC)— which is reserved by the firmware during the boot process. This memory range is managed by the untrusted operating system, while its contents are encrypted by the CPU package with a random key. Currently, SGX supports a maximum of 128 MB of EPC for all enclaves running on a system together. Memory demand exceeding this range requires the SGX driver to re-encrypt enclave pages and copy them to regular system memory causing a high performance impact.

3 Design

With our *Vert.x Vault* we want to integrate TEEs into Vert.x applications such that the TEE developer can offload parts of the application logic to SGX enclaves. This allows usage of SGX for critical parts of the application logic while keeping

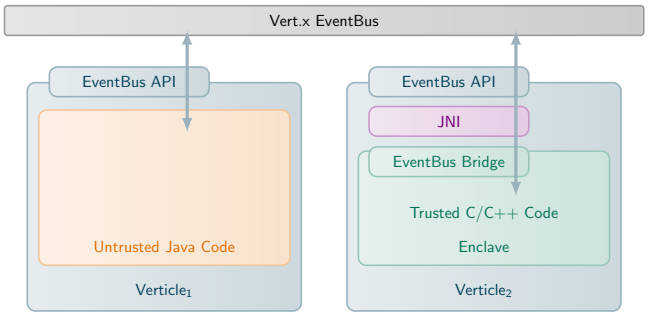

Fig. 3: Architectural Overview of *Vert.x Vault* showing two verticles connected via the Vert.x event bus and one of them containing an SGX enclave.

most of it including the code base of Vert.x untrusted and not part of the TCB. Vert.x and SGX TEEs are a suitable fit, as the micro service paradigm matches the original idea of the SGX engineers to keep enclaves lightweight and small, and the ability of SGX to integrate multiple TEEs into a single user process.

Since Vert.x is written in Java and enclaves are written in C/C++, we propose integration of enclaves via Java Native Interface (JNI) into verticles. While the integration of a full-fledged JVM into an enclave would be possible, it is not favourable as it introduces a lot of complexity to the enclave and causes a drastic increase of the TCB, leading to a higher probability of exploitable security vulnerabilities and an increased attack surface. Furthermore, in the context of SGX this would lead to a significant performance penalty as the available enclave memory—limited to 128 MB on current hardware—would be exceeded. Exceeding the enclave memory causes the SGX driver to move enclave pages to regular system memory, requiring an expensive re-encryption of the pages [8].

An overview of our architecture is shown in Figure 3, illustrating two verticles connected via the Vert.x event bus, and one of them containing an SGX enclave. As can be seen from the figure, a *secure verticle* is a verticle with an integrated enclave, that can be reached by other verticles via the Vert.x event bus. This enables the developer to design large micro service applications consisting of a mix of multiple untrusted and secure verticles. It is also possible to explicitly guide further isolation via process boundaries, as verticles can be deployed on specific hosts and multiple verticles can reside in the same JVM process.

The enclave which is integrated into a secure verticle, contains our Vert.x Vault as a C/C++ library offering an Application Progamming Interface (API) to access the event bus directly from the enclave. In order to integrate the C/C++-based enclave into the Java-based Vert.x tool-kit, we use a small JNI component to forward the *Vert.x Vault* calls to the original Vert.x API. By this, the enclave is able to register an address on the event bus and receive messages from other verticles, as well as sending messages to arbitrary addresses on the event bus.

3.1 Adversary Model and Assumptions

By using SGX to implement TEEs we gain a very high level of security allowing a very strong adversary model. We assume an attacker that has full access to the machines running Vert.x, including the firmware, Operating System (OS), and all system and user space software. Even physical attacks such as cold boot attacks can be allowed without violating the security guarantees that SGX provides. We assume a correct implementation of SGX and all cryptographic primitives.

3.2 Security Aspects

As already outlined, the size of the TCB influences the overall security of an application. Besides its complexity and the fact that syscalls can not be done in an enclave, this is the main security reason why a full-blown JVM should not be ported to run inside an enclave. Running a JVM inside an enclave would lead to both, a large TCB inside the enclave as well as a large memory footprint of the enclaves which negatively affects both, security and performance.

Hence, in our approach we aim at C/C++-based enclaves that execute only minimal parts of the application logic in the trusted environment. Partitioning the application logic to decide which parts of application logic are required to run inside the enclave and which not is the task of the developer. We demonstrated this approach—called *application partitioning approach*— for ZooKeeper, a complex Java-based coordination service [8], in an earlier work.

Running parts of application logic in an enclave, provides protection of confidentiality and integrity of data inside the enclave and the enclave code itself. However, once data is exchanged between and enclave and the untrusted outside world, data must also be protected in an adequate way; traditionally by using using TLS between the two parties communicating. However, in case of SGX, the TLS endpoint must reside inside the enclave. In our *Vert.x Vault*, the enclave integrated into a secure verticle inherits the Intel SGX SDK features and can resort to the included cryptographic functions in order to protect data exchanged via the Vert.x event bus or with other secure verticles, respectively.

Enclave Interface Another possible security aspect is the enclave interface, i.e. the amount and signature of ecalls. In general, there will be less security vulnerabilities with a more narrow enclave interface. Also, we wanted to keep the enclave interface generic in order to support any kind of application logic running in the enclave. The complete enclave interface is illustrated in Listing 1.1.

We defined an enclave entry method which is called after the enclave is created. This allows the implementation of initialisation routines inside the enclave and to register to event bus addresses on secure verticle start-up and is analogous to the verticle's constructor method. Next, we implemented another method to deliver messages from the event bus to the enclave. Only within the enclave we distinguish the address on the event bus and deliver the message to the respective callback function.

```
enclave {
  trusted {
    public void ecall_enclaveInit();
    public void ecall_deliverMsg(
      [in, size=lenCh] const char *channel, size_t lenCh,
      [in, size=lenMsg] const char *msg, size_t lenMsg);
  };
  untrusted {
    void ocall_register( [user_check] void* weak,
      [in, string] const char *channel, size_t len);
    void ocall_unregister( [user_check] void* weak,
      [in, string] const char *channel, size_t len);
    void ocall_send( [user_check] void* weak,
      [in, string] const char *channel, size_t len,
      [in, string] const char *msg,   size_t lenMsg);
    void ocall_broadcast( [user_check] void* weak,
      [in, string] const char *channel, size_t len,
      [in, string] const char *msg, size_t lenMsg);
  };
};
```

Listing 1.1: Enclave interface description.

Calls from the enclave to the untrusted context are also part of the enclave interface. Firstly, there are two methods for registering and de-registering an address on the Vert.x event bus. During registration, the enclave stores a function pointer in an internal (trusted) data structure to the user-defined callback function once a message is received for this address. In order to allow sending messages to the event bus, we implemented two ocalls to send and broadcast messages onto the event bus respectively.

3.3 Programming Model

A crucial requirement of our project was to give the enclave developer the impression of the Vert.x programming model inside an enclave. Consequently, we defined the notion of a *secure verticle* that essentially represents a verticle whose application logic is implemented inside an enclave. As a Vert.x application will usually consist of multiple verticles, the idea is to offload all sensitive application logic components to secure verticles that can interact with the other verticles via the event bus. In terms of the application partitioning approach from our earlier work [8], only application logic fragments that require direct plain text access to user data should be implemented as secure verticles, whereas all other parts should be untrusted verticles. This is compliant with our goal of minimising the TCB for a higher level of security.

Inside enclaves of a secure verticle, we want to give the developer a Vert.x-like programming model, i.e. reactive event-based callbacks. For this purpose, we mimic the Vert.x event bus API inside the enclave, and allow the enclave developer to register event bus addresses and receive callbacks for the registered addresses as well as sending messages to the event bus. By this, the secure verticles get integrated into the complete system of verticles of a full Vert.x application.

4 Enclave Integration and *Vert.x Vault* Features

The integration of SGX enclaves into Java applications in general, and into the Vert.x tool-kit specifically in this work, can be done using JNI. For this purpose we implement the interface that the enclave uses to interact with the event bus, as well as the one that Vert.x uses to forward incoming events to the enclave in JNI. In conjunction with stub code generated by the SGX Software Development Kit (SDK) supporting interaction of untrusted C/C++ code and the enclave code, this requires additional copies of the message buffers. We discuss their performance impact in our evaluation in Section 6.

In order to support multiple secure verticles coexisting in the Vert.x environment, we need to maintain an enclave identifier in the Java class representing the secure verticle, such that each secure verticle "knows" its associated enclave.

Furthermore, we want to enable one secure verticle to register multiple individual callback functions inside the enclave for different event bus addresses. This requires to maintain a distinct entry point inside the enclave for all incoming messages where a lookup happens in order to find the right callback function for this event. Including the deregistration of addresses on the event bus, all these features are implemented as part of our *Vert.x Vault*.

4.1 Bootstrapping and Remote Attestation

SGX enclaves inherently are not able to contain secrets when they are created, thus injection of secrets into enclaves must be done after a successful Remote Attestation (RA). Integration of RA into our *Vert.x Vault* architecture requires a generic untrusted component inside of each secure verticle that can create a "quote" of the enclave and publish it on the event bus to an administrator component. Along with a public key generated by the enclave, the quote can be used by an administrator to verify the secure enclave remotely and via the Vert.x event bus by using the Intel attestation service [4]. After a successful RA, the administrator can inject secrets into the enclaves of secure verticles.

4.2 Secure Event Bus

Recently, the Vert.x developers added the *secure event bus* to Vert.x, that allows the encryption of all messages on the event bus transparently to the verticles. Essentially, the secure event bus provides TLS-based transport security of messages across machine boundaries. In order to be transparent to verticles, the integration of the secure event bus is done deeply inside Vert.x. Hence, the verticles can not notice the encryption or—in case of a secure verticle using our *Vert.x Vault*—move the encryption endpoints into the enclaves. For this reason, we consider the secure event bus orthogonal to the implementation of our *Vert.x Vault*. In order to integrate it with our secure verticles, the TLS endpoint of the secure event bus must be moved inside the enclave, requiring major changes to the internals of Vert.x. However, even without integration of the secure event bus with secure verticles, the latter can of course establish their own secure connections transparently to the event bus.

5 Use Case Scenario SERECA Project

In the context of the SERECA project, we demonstrate the opportunities coming from an integration of SGX with Vert.x through a suitable use case scenario: a cloud-based application for the monitoring of seven dams belonging to a water supply network [9]. This use case unarguably belongs to the Critical Infrastructure (CI) domain. CIs enclose assets essential for the functioning of all countries' fundamental facilities such as energy, telecommunications, water supply, and transport. Due to their importance in nations' sustainability, CIs are target of terrorist cyber-attacks, as demonstrated by recent events: "Black Energy"[4] in 2015, and "Havex"[5] in 2014. Security threats related to the disclosure or manipulation of sensitive data slows the migration of CIs to cloud technologies.

Through the cloud-based Water Supply Network Monitoring (WSNM) application, we want to prove that the combination of Vert.x with SGX—enabled by our *Vert.x Vault*—can represent a possibility to overcome CI's security concerns. The different tools provided by Vert.x enable the development of a reactive WSNM application based on micro-services having the following peculiarities: 1) Easily deployable among the dams and the cloud; 2) Highly scalable in front of sensors measurements peaks 3) Highly available in front of failures; 4) Highly performant in the process of sensors data collection, elaboration and provision. Moreover, the SGX extension allows the encryption/decryption and sealing of sensitive measurements by an *enclave* using a platform-specific key and while requiring to trust only the CPU package.

Figure 4 shows the overall architecture of the WSNM distributed application. On the dam-side, a *data collector verticle* interfaces, through a ModBus protocol, with a data logger equipment responsible for providing all the sensor data. Then, the acquired measurements are sent to the registered cloud-*verticles* through the Vert.x *secure event bus*. Thanks to this, messages are securely encrypted and signed. Only the involved receiver verticles, knowing the encryption key (AES) and integrity key (HMAC), can decrypt the message. In addition to a TLS secure communication, a *Route-based* encryption is enforced. Senders and receivers do not indicate the key in the message. They use an equivalent configuration, which defines the key to be used for a specific address. In this way, they do not share key names, but agree on the address-key mapping.

On the cloud-side, four registered *verticles* receive data through the secure event bus and, based on their duties, take a specific action on it. Two of them are *secure verticles* and so make use of *Vert.x Vault* APIs seen in Section 3.3:

Cache Archiver Verticle (CAV): It is responsible for the storage of time-recent data into an in-memory system. Such a data is needed, e.g., by the alarm manager in charge of real-time analysis activities. The CAV is unarguably a *Secure Verticle* as the measurements it stores in memory must be encrypted. Therefore, the CAV registers itself from within the enclave to the Vert.x secure event bus through a *ocall_register*. Then, when new updates are received, the data—before being

[4] `https://ics-cert.us-cert.gov/alerts/ICS-ALERT-14-281-01B`
[5] `https://ics-cert.us-cert.gov/alerts/ICS-ALERT-14-176-02A`

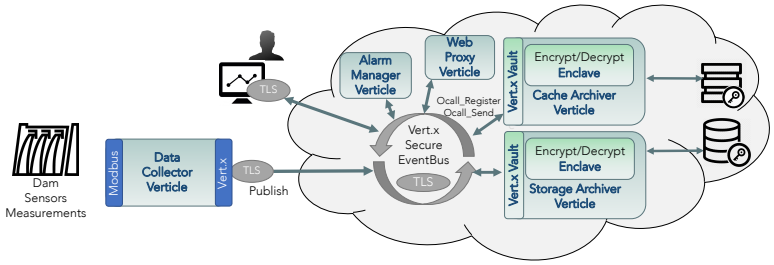

Fig. 4: Architecture of the dam monitoring application SGX-enabled

stored—is encrypted into the enclave. It can obviously happen that other *verticles* (e.g. the AMV) asks for that data. When this happens, a *ocall_send* is performed to provide the measurements to the interested verticle.

Alarm Manager Verticle (AMV): The alarm manager is in charge of signalling dangerous situations occurring on the dam infrastructure by enforcing a *Complex Event Processing (CEP)*, i.e., correlating different data sources to find events or patterns suggesting more complicated circumstances. To do that, the AMV needs to receive live data from the collector verticle and also temporarily store data in memory. For this reason, it communicates with the CAV through point-to-point messages exchanged through the secure event bus. The AMV, that currently operates on data securely stored, will be extended as a *secure verticle* to realize the CEP processing into an *enclave*.

Storage Archiver Verticle (SAV): Beside saving data for CEP processing purposes, it is also important to store measurements for historical trend evaluations into a persistent storage system. Even in this case it is important that stored data is encrypted. Hence, even the SAV is defined as a *Secure Verticle*.

Web Proxy Verticle (WPV): The final user operator has access to the monitoring application through a web-based dashboard, which reports real-time measurements, historical trends, and alarm notifications. Such a dashboard communicates with the back-end cloud application through the WPV, which can ask any verticle for data to be sent to the web browser based on the requests. The TLS-enabled transmission of sensitive data is realized through a Vert.x *secured SockJS bridge*—provided in the *secure event bus*—able to encrypt/decrypt data at application level using a key shared with the web browser.

6 Evaluation

In order to measure the performance impact of enclaves integrated into verticles using our *Vert.x Vault*, we wrote a benchmark tool in order to evaluate the performance of our prototype and present the results in this section. All measurements presented in this section were done with real enclaves and executed on SGX-capable machines with Core i7-6700 @3.4GHz, 24 GB RAM, 256 GB SSD and 4x GbE.

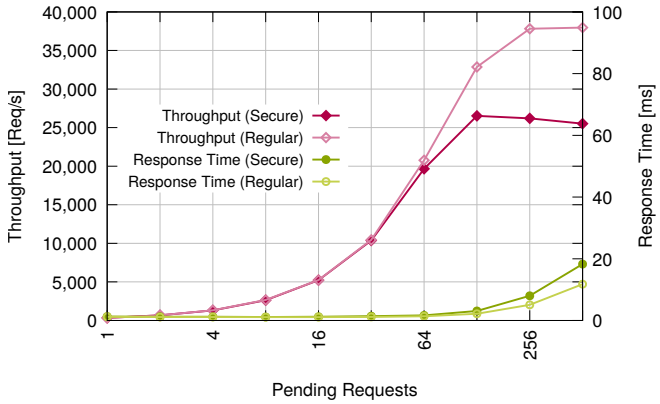

Fig. 5: Request throughput for various number of pending requests.

6.1 Performance Measurement

The measurement scenario comprises three verticles deployed on two hosts: a sender verticle that sends requests and measures the response time and throughput on one host, and a regular and a secure verticle both running on the other host answering the requests. All communication between verticles is transferred via the Vert.x event bus using the Hazelcast cluster manager[6], which is the default cluster manager implementation of Vert.x. In order to get an impression of the performance impact of the usage of SGX in verticles, we exchange payloads of various size between the verticles and measure throughput and response time.

In a first experiment we investigate the level of concurrency that leads to optimal throughput by increasing the maximum allowed number of concurrent pending requests on the event bus in the sender verticle. We chose a realistic payload size of real Vert.x applications of 1024 Bytes for this experiment. The results of this experiment are shown in Figure 5 which also includes the response times. As can be seen, the value of 128 pending requests leads to the highest throughput, while higher values do not significantly increase throughput.

We also measure the throughput of regular and secure verticles for various sizes of message payload using the aforementioned value of 128 concurrently pending requests that led to optimal throughput. Figure 6 illustrates the request throughput of our experiment for various payload sizes. It proves that secure verticles perform quite well, and in most of the cases reach a throughput almost as high as regular verticles. The same is true for the transmitted payload traffic on the event bus between the verticles, as can be seen from the same Figure 6.

Finally, we measured the mean response time of requests for regular and secure verticles as illustrated in Figure 7. The figure shows the additional processing time required to enter the enclave and copy the payload buffer in and out. As expected, in general the response time of secure verticles is higher than the one of regular

[6] `https://hazelcast.com/`

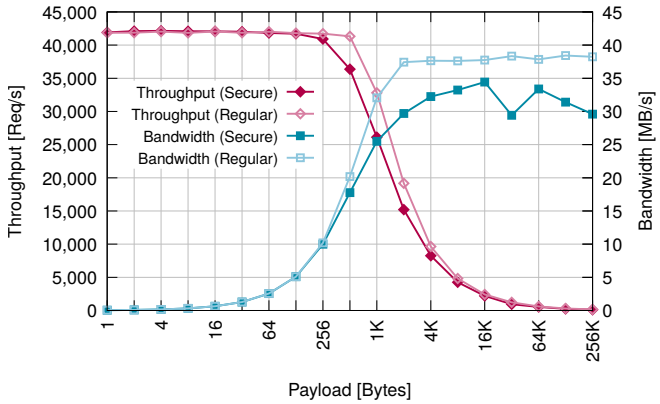

Fig. 6: Request throughput and traffic for various payload sizes.

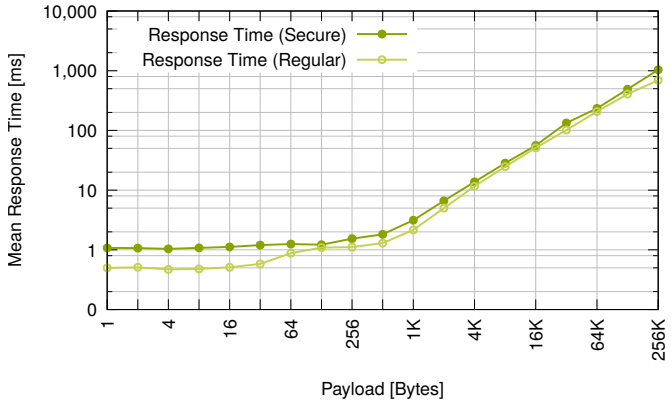

Fig. 7: Response times for various payload sizes.

verticles. For very small payloads between $0 - 512$ Bytes there is a *relatively* high overhead of $\approx 83.1\%$ of secure verticles, while for larger payloads between 512 Bytes and 256 KBytes the relative overhead decreases to only $\approx 26.2\%$. We explain this by the constant overhead of entering and exiting the enclave which is relatively more notable for small payloads.

6.2 Size of TCB

As motivated earlier in this paper, we aimed at minimising the Source Line of Code (SLOC) inside the enclave (i.e. the TCB) in order to optimise both performance and security. While the Vert.x tool-kit's code base comprises thousands of SLOC, our *Vert.x Vault* only requires 69 lines of code inside the enclave in order to enable the enclave code to access the Vert.x event bus. Apart from this, the TCB

naturally comprises the code by the micro service application developer, but the majority of code resides outside of the enclave and untrusted. This enables the developer to decide on a very fine-grained level what code is trusted.

7 Related Work

Haven [6] was the first system and supports execution of legacy applications unchanged in an SGX enclave. While running legacy applications in enclaves has many benefits, such as support for closed source applications, this feature adds a large amount of code to the TCB of the enclave, like a full library OS and other libraries. This eventually leads to a large memory footprint, which negatively influences the performance of the application on top of SGX.

SCONE [5] supports running secure containers based on Docker [1]. This approach reduces the memory footprint of the enclaves when compared to Haven, while still featuring a generic platform for the execution of secure containers.

Another type of related work is the field of partitioned applications, that are specifically paritioned applications to run most efficiently and with the least possible memory footprint on top of SGX. An example for securing complex cloud services by partitioning them for usage with SGX enclaves is SecureKeeper [8,7] featuring a partitioned Apache ZooKeeper [10] coordination service.

The communication of verticles on the Vert.x event bus is alike publish-subscribe systems. In this context, Pires et al. presented secure content-based routing with SGX [14]. Their system protects the subscription process of clients at data providers and the confidentiality of the data exchanged between them. In contrast, our *Vert.x Vault* not only enables protecting the confidentiality of exchanged messages between verticles, but also allows general-purpose sensitive data processing of inside secure verticles.

8 Conclusion

Micro service architectures are a modern way of writing scalable applications tailored for cloud environments. However, trust issues still hinder the adoption of cloud technology especially for sensitive applications handling sensitive data.

In this work, we showed an approach of integrating TEEs into the micro service tool-kit Vert.x. With our approach due to the usage of SGX, we gain a high level of security at a strong adversary model. Furthermore, we support the Vert.x programming model and control flow inside the TEE by mimicing its API. This allows interaction of trusted code with the Vert.x event bus.

Our prototype implementation demonstrates the feasibility of our approach, while the evaluation shows a very low performance overhead of only $\approx 8.7\%$.

9 Acknowledgements

This project received funding from the European Union's Horizon 2020 research and innovation programme under the SERECA project (Grand No. 645011).

References

1. Docker. `https://www.docker.com/`, last accessed 02/01/2017.
2. Eclipse Vert.x. `http://vertx.io/`, last accessed 01/25/2017.
3. What Led Amazon to its Own Microservices Architecture. `http://thenewstack.io/led-amazon-microservices-architecture/`, last accessed 01/25/2017.
4. I. Anati, S. Gueron, S. Johnson, and V. 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*, 2013.
5. S. Arnautov, B. Trach, F. Gregor, and T. Knauth. SCONE: Secure linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.
6. A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.
7. S. Brenner, C. Wulf, and R. Kapitza. Running ZooKeeper Coordination Services in Untrusted Clouds. In *10th Workshop on Hot Topics in System Dependability*, 2014.
8. S. Brenner, C. Wulf, M. Lorenz, N. Weichbrodt, D. Goltzsche, C. Fetzer, P. Pietzuch, and R. Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 15th International Middleware Conference*, 2016.
9. G. Cerullo, G. Mazzeo, G. Papale, L. Sgaglione, and R. Cristaldi. A secure cloud-based SCADA application: The use case of a water supply network. In *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Fifteenth SoMeT_16, Larnaca, Cyprus, 12-14 September 2016*, pages 291–301, 2016.
10. P. Hunt, M. Konar, F. Junqueira, and B. Reed. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 2010.
11. K. R. Jayaram, D. Safford, U. Sharma, V. Naik, D. Pendarakis, and S. Tao. Trustworthy Geographically Fenced Hybrid Clouds. In *Proceedings of the 15th International Middleware Conference*, 2014.
12. 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*, 2013.
13. S. Pearson and A. Benameur. Privacy, Security and Trust Issues Arising from Cloud Computing. In *IEEE 2nd International Conference on Cloud Computing Technology and Science*, 2010.
14. R. Pires, M. Pasin, P. Felber, and C. Fetzer. Secure Content-Based Routing Using Intel Software Guard Extensions. In *Proceedings of the 17th International Middleware Conference*, 2016.
15. Synopsys, Inc. Open Source Report 2014. `http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf`, 2014.