Montsalvat: Intel SGX shielding for GraalVM native images

Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux, Jean-Pierre 2021 Lozi

Montsalvat: Intel SGX Shielding for GraalVM Native Images

Peterson Yuhala
University of Neuchâtel
Neuchâtel, Switzerland
peterson.yuhala@unine.ch

Jämes Ménétrey
University of Neuchâtel
Neuchâtel, Switzerland
james.menetrey@unine.ch

Pascal Felber
University of Neuchâtel
Neuchâtel, Switzerland
pascal.felber@unine.ch

Valerio Schiavoni
University of Neuchâtel
Neuchâtel, Switzerland
valerio.schiavoni@unine.ch

Alain Tchana
ENS
Lyon, France
alain.tchana@ens-lyon.fr

Gaël Thomas
Télécom SudParis
Institut Polytechnique de Paris
France
gael.thomas@telecom-sudparis.eu

Hugo Guiroux
Oracle Labs
Zürich, Switzerland
hugo.guiroux@oracle.com

Jean-Pierre Lozi
Oracle Labs
Zürich, Switzerland
jean-pierre.lozi@oracle.com

ABSTRACT

The popularity of the Java programming language has led to its wide adoption in cloud computing infrastructures. However, Java applications running in untrusted clouds are vulnerable to various forms of privileged attacks. The emergence of trusted execution environments (TEEs) such as Intel SGX mitigates this problem. TEEs protect code and data in secure *enclaves* inaccessible to untrusted software, including the kernel and hypervisors. To efficiently use TEEs, developers must manually partition their applications into trusted and untrusted parts, in order to reduce the size of the trusted computing base (TCB) and minimise the risks of security vulnerabilities. However, partitioning applications poses two important challenges: *(i)* ensuring efficient object communication between the partitioned components, and *(ii)* ensuring the consistency of garbage collection between the parts, especially with memory-managed languages such as Java. We present Montsalvat, a tool which provides a practical and intuitive annotation-based partitioning approach for Java applications destined for secure enclaves. Montsalvat provides an RMI-like mechanism to ensure inter-object communication, as well as consistent garbage collection across the partitioned components. We implement Montsalvat with GraalVM native-image, a tool for compiling Java applications ahead-of-time into standalone native executables that do not require a JVM at runtime. Our extensive evaluation with micro- and macro-benchmarks shows our partitioning approach to boost performance in real-world applications up to 6.6× (PalDB) and 2.2× (GraphChi) as compared to solutions that naively include the entire applications in the enclave.

CCS CONCEPTS

• **Security and privacy** → **Trusted computing**; • **Software and its engineering** → *Object oriented languages.*

KEYWORDS

Trusted Execution Environments, Intel SGX, Managed Execution Environments, Java, GraalVM

1 INTRODUCTION

The Java programming language is widely used in cloud infrastructures. Popular cloud frameworks such as Hadoop [19], ZooKeeper [21] and Spark [20] are based on Java. The recent growth of cloud-based services surrounding these popular tools raises security and privacy concerns. To address security issues in the cloud, major CPU vendors have introduced trusted execution environments (TEEs), such as Intel SGX [16], AMD SME [27] and ARM TrustZone [3], which shield sensitive code and data inside secure memory regions called *enclaves*. In spite of their attractive security properties, programming TEEs is complex: it is usually done in compiled languages and low-level APIs at the function level, and it requires developers to make non-trivial efforts to minimise the trusted computing base (TCB). Enforcing privacy with TEEs in a high-level, managed language like Java is particularly challenging.

Solutions exist to run entire applications (including the JVM) inside enclaves, while relying on a library OS [4, 7, 40, 52] to emulate unsupported OS logic in the enclave. This approach offers good compatibility for legacy applications and requires little intervention from developers. However, it significantly increases the size of the TCB: library OSs inside the enclave typically hit millions of lines of code [47], which violates the principle of least privilege [43] and increases the chances of enclave vulnerabilities.

Others (*e.g.*, Civet [53], Uranus [26]) try to mitigate this problem by partitioning Java applications for enclaves. Civet leverages static analysis [6] to partition Java applications, but embeds a JVM and a library OS [52] inside the enclave, hence resulting in a large TCB. Uranus provides a technique to partition Java applications by annotating sensitive methods, but it requires developers to use third-party tools to infer trusted partitions of applications, as well as manual intervention by developers during the partitioning process. One can manually partition specific Java frameworks for enclaves [9, 44, 60], but this approach cannot be used for generic applications. Some systems like Glamdring [30] propose techniques to partition

native applications in C and C++ automatically, but they cannot be used for Java applications that rely on a managed runtime.

Partitioning Java applications for enclaves raises two significant challenges which were not sufficiently addressed by previous work:

(1) Code running outside of an enclave (in the untrusted runtime) may allocate objects inside the enclave (in the trusted runtime), and code running inside the enclave may allocate objects outside of the enclave. Since both runtimes operate on separate memory heaps, there is a need for an efficient mechanism to ensure object communication across the two runtimes.
(2) Since there may be references between the untrusted runtime and the enclave, the garbage collector have to be extended to ensure consistency, *i.e.*, objects in one runtime should not be destroyed if objects in the opposite runtime still reference them.

To address these problems, we propose Montsalvat, a tool that leverages annotations to partition Java applications into trusted and untrusted components automatically. Montsalvat leverages bytecode transformation to split Java applications between the trusted and untrusted runtimes, and applies distributed techniques like remote method invocation [22] to enable efficient communication between trusted and untrusted objects. Montsalvat introduces a dedicated GC helper to synchronise garbage collection (GC) between both runtimes. Contrary to approaches based on library OSs [40, 52, 53], Montsalvat introduces a shim library in the enclave that relays unsupported libc calls in the enclave to the untrusted runtime, which reduces the TCB.

We implemented our approach with GraalVM [8] and Intel SGX. GraalVM is a high-performance JDK distribution that makes it possible to build and run applications implemented in a wide range of high-level languages (*e.g.*, Java, Scala, JavaScript, Clojure, Kotlin, *etc.*). Montsalvat leverages a GraalVM tool named *Native Image* supporting ahead-of-time (AoT) compilation of applications into native executables, called *native images* [18, 35, 57], which do not require a JVM at runtime. GraalVM native-image AoT compiles only reachable application methods, classes and fields, thereby excluding any redundant application logic from the final executable. This results in quicker startup times and lower memory footprint for applications. These properties of GraalVM native images make them particularly well-suited for restricted environments such as enclaves. We evaluate Montsalvat using synthetic benchmarks as well as real-world Java applications such as LinkedIn's PalDB [31] and GraphChi [29]. Our evaluation results show that partitioning PalDB and GraphChi can yield up to 6.6× and 2.2× performance improvements respectively, as compared to solutions that run these applications on a JVM in the enclave.

In summary, we propose the following contributions:

- A practical and intuitive annotation-based approach for partitioning Java applications into insecure classes and secure classes to be run inside TEE enclaves.
- An RMI-like mechanism for inter-object communication across the trusted and untrusted runtimes.
- A garbage collection extension to synchronise object destruction across the trusted and untrusted heaps.
- Extensive experimental evaluation with various applications demonstrating the efficiency of our approach.

This paper is organised as follows. §2 provides background concepts and §3 discusses related work. Our threat model is introduced in §4. The architecture and implementation of Montsalvat are detailed in §5. §6 presents our extensive experimental evaluation, and we conclude in §7.

2 BACKGROUND

2.1 Intel software guard extensions

Intel software guard extensions (SGX) is an extension to the Intel instruction set architecture [46] that enables applications to create *enclaves*, *i.e.*, secure isolated regions in memory. Enclave code and data are stored in a secure memory region, the enclave page cache (EPC). All EPC pages in DRAM are encrypted and only decrypted by a memory encryption engine (MEE) when they are loaded into a CPU cache line. Recent Intel processors support a maximum of 256 MB of EPC memory (only 192 MB are usable by SGX enclaves), limiting the amount of data in the enclave at any given time. The Linux SGX kernel driver can swap pages between the EPC and regular DRAM. This paging mechanism lets enclave applications use more than the total EPC, but at a significant cost [9, 50].

SGX applications are typically partitioned into *trusted* and *untrusted* parts that handle sensitive and non-sensitive operations, respectively. Enclaves only run in user mode [16], hence OS services such as system calls cannot be executed directly inside of them and are instead relayed to the untrusted runtime. To enable communication across runtimes, Intel SGX provides `ecall` and `ocall` routines, which are specialised function calls that are used to respectively enter and exit an enclave. These calls induce costly context switches that last up to 13,100 CPU cycles [55, 59].

The Intel SDK [15] makes it possible to partition and build enclave-based applications in C/C++ manually. The SDK provides an enclave definition language (EDL) to define the enclave's interface, *i.e.*, the set of all `ecall` and `ocall` routines. An additional tool, *Edger8r*, generates *edge routines* using the EDL specifications. The edge routines sanitise and marshal data into and out of the enclave. All enclave code is then compiled into a final shared object that is cryptographically hashed for verification at runtime when it is loaded into enclave memory.

Manual partitioning can be avoided as solutions exist to easily run entire applications inside enclaves (*e.g.*, SCONE [4], Graphene-SGX [52], SGX-LKL [40], *etc.*). However, these solutions have large TCBs, which degrade application performance and increase the chances of enclave vulnerabilities.

2.2 GraalVM native-image

GraalVM native-image is a tool, built on top of the GraalVM compiler [8], to compile ahead-of-time applications into standalone executables, which are named *native images*. It supports JVM-based languages, *e.g.*, Java, Scala, Clojure and Kotlin. Native images can also execute dynamic languages such as JavaScript, Ruby, R or Python [35]. GraalVM native-image leverages a points-to analysis [5, 42, 57] approach to find all the reachable application methods that are compiled into the final native image, leading to faster startup times and lower memory footprint as compared to other Java runtimes. GraalVM native-image enables applications to execute initialisation code (*e.g.*, reading and parsing a configuration file) at build time, effectively reducing the application startup as less logic is executed at run time. To transfer the result of the initialisation (Java objects) from build to runtime, GraalVM native-image

takes a snapshot of the heap (called the *image heap*) at the end of the build, and stores it into the generated executable. The image heap is memory mapped inside the application heap at startup, allowing the application to start from the state initialised at build time.

GraalVM native-image makes a closed-world assumption, *i.e.*, it considers that all application classes that can be executed at run time are known and available at build time. To support dynamic features such as reflection, the user provides a list of the classes, fields, and methods that can be accessed dynamically. Each element of this list is then always included in the native image, in addition to all classes, fields and methods transitively reachable from these elements. This list can be provided through e.g., CLI options, programmatically, or a JSON file. GraalVM native-image provides a *tracing agent* [39] which assists developers in generating such a JSON file.

GraalVM native images do not run on a regular JVM (*e.g.*, HotSpot): runtime components that are needed to run JVM-based applications, such as a garbage collector, support for thread scheduling and synchronisation, as well as stack walking and exception handling are directly included inside the created native images.

GraalVM native-image provides the possibility of creating multiple independent VM instances at runtime, which are called *isolates*. Each isolate operates on a separate heap, allowing garbage collection to be performed independently. Thus, threads executing in one isolate are not affected by garbage collection done in another isolate. Montsalvat creates a default isolate for each one of the two runtimes of the partitioned Java application (trusted and untrusted), which provides the execution contexts for all "entry point" methods (*e.g.*, `main`).

3 RELATED WORK

We classify the related work into four categories: *(i)* systems that make it possible to run full, unmodified applications inside enclaves, *(ii)* framework-specific systems that support partial execution inside enclaves, *(iii)* systems that allow for partitioning generic native applications, and *(iv)* systems that allow for partitioning generic Java applications.

Running full applications inside enclaves. Prior systems such as Haven [7], SCONE [4], Graphene-SGX [52] and SGX-LKL [40] propose solutions to run entire legacy applications inside enclaves. They introduce a library OS into the enclave to emulate OS logic. For instance, SCONE leverages a modified version of the *libc* to run microservices inside Docker [17] containers. While these solutions offer good compatibility with a wide range of applications and require low developer effort, they introduce millions of lines of code into the TCB. This may significantly decrease enclave performance and leaves more room for security vulnerabilities.

Framework-specific partitioning. Some recent systems propose to manually partition specific frameworks and/or the applications that run on them into trusted and untrusted parts. VC3 [44] is a system for trustworthy data analytics in Hadoop that requires manually rewriting Map and Reduce functions to be used in SGX enclaves, while keeping the main Hadoop library outside the enclave. SecureKeeper [9] proposes an extension to ZooKeeper which preserves confidential user data inside enclaves while maintaining the ZooKeeper framework outside the enclave. The authors of Plinius [59] manually partition a persistent memory and machine learning (ML) library in order to enable efficient ML in SGX enclaves. Opaque [60] is a secure data analytics platform built on top of Spark SQL that focuses on preventing access pattern leakage; it notably introduces SGX-enabled oblivious operators that can be used on tables that store sensitive data. These systems help reduce the size of the TCB but focus on individual frameworks, which limits their flexibility.

Native code partitioning. Glamdring [30] provides a technique to automatically partition C/C++ applications into untrusted and trusted parts using static program slicing. Panoply [47] introduces *micro-containers* which expose standard POSIX abstractions and run inside enclaves; applications must be refactored to extract sensitive code and data to be placed inside micro-containers. These systems do not tackle the complexities introduced by managed languages.

Java code partitioning. Civet [53] and Uranus [26] are two recent frameworks for running parts of Java applications inside enclaves. Civet requires defining methods which will serve as the partitioning boundary, and Uranus requires annotating all sensitive methods. We argue that placing the enclave boundary at class level is more intuitive for developers. Additionally, our system benefits from GraalVM's optimisations, such as class initialisations at build time. Rather than a small shim library that relays unsupported calls to the trusted runtime as in our approach, Civet stores a full library OS inside the enclave (specifically, it uses Graphene SGX [52]), which leads to a larger trusted code base. CFHider [54] also proposes to run parts of Java applications inside enclaves, but it specifically focuses on branch statement conditions with the objective to guarantee control flow confidentiality.

4 THREAT MODEL

Montsalvat assumes enclave code and the CPU package are trusted, similar to related work with SGX [4, 26, 30, 44, 53]. The source code annotation, image build via AoT compilation, and final enclave signing are done in a trusted environment. This prevents malicious classes/bytecode from being introduced into the enclave at runtime [32]. The integrity of the enclave can then be validated at runtime via remote attestation [12, 16] mechanisms.

Montsalvat supports a powerful adversary with control over the full software stack, including the OS, hypervisors, and access to the physical hardware (*e.g.*, DRAM, secondary storage, *etc.*). The adversary's goal is to gain access to confidential data (*e.g.*, passwords, encryption keys, *etc.*) which may be processed in trusted application classes, or to damage the integrity of confidential data.

Montsalvat is resilient to physical attacks like cold boot attacks [24] aimed at reading sensitive data in DRAM, or bus probing [58] to read the memory channel between the CPU and DRAM; the SGX security model [16] prevents these.

We assume the adversary cannot physically open and manipulate the SGX-enabled processor package (as in [13]), and that the enclave code does not intentionally leak sensitive data. Denial-of-service and side-channel attacks [10, 45], for which mitigations exist [23, 33], are considered out of scope.

5 ARCHITECTURE

The main goal of Montsalvat is to partition Java applications for SGX enclaves. The final partitioned application includes a *trusted* and an *untrusted* part, respectively running inside and outside the

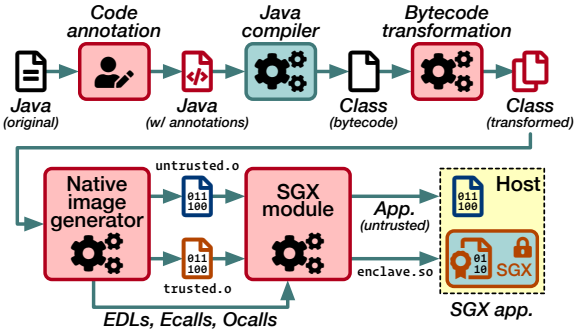

Figure 1: Overview of Montsalvat's workflow. Annotated Java code, once compiled, is processed by the bytecode transfomer to produce two versions of the application classes. These classes are then used by the native image generator to produce trusted and untrusted images, which are finally linked with enclave libraries in the SGX module to build the final SGX application.

```
1  @Trusted
2  public class Account {
3    private String owner;
4    private int balance;
5    public Account(String s, int b) {
6      this.owner = s;
7      this.balance = b;
8    }
9    public void updateBalance(int v) {
10     this.balance += v;
11   }
12 }

13 @Trusted
14 public class AccountRegistry {
15   private List<Account> reg =
16     new ArrayList<Account>();
           // Trusted in trusted obj
17   public AccountRegistry() {}
18   public void addAccount(Account a) {
19     this.reg.add(a);
20   }
21 }

22 @Untrusted
23 public class Person {
24   private String name;
25   private Account account;
           // Trusted in untrusted obj
26   public Person(String s, int v) {
27     this.name = s;
28     this.account = new Account(s, v);
29   }
30   public Account getAccount() {
31     return this.account;
32   }
33   public void transfer(Person p, int v) {
34     p.getAccount().updateBalance(v);
35     this.account.updateBalance(−v);
36   }
37 }

38 @Untrusted
39 public class Main {
40   public static void main(String[] args) {
41     Person p1 = new Person("Alice", 100);
42     Person p2 = new Person("Bob", 25);
43     p1.transfer(p2, 25);
44     AccountRegistry reg =
45       new AccountRegistry();
46     reg.addAccount(p1.getAccount());
47   }
48 }
```

Listing 1: Illustrative example with annotated classes using Montsalvat: trusted classes (lines 1-12 and 13-21), and untrusted (lines 22-37 and 38-47).

enclave. Figure 1 depicts Montsalvat's complete workflow, from the source code to the generation of the final SGX application. It comprises 4 main phases: *(1)* code annotation, *(2)* bytecode transformation, *(3)* native image partitioning, and *(4)* SGX application creation.

To illustrate the inner workings of Montsalvat, we consider a synthetic Java application to be partitioned (see Listing 1). Three classes mutually interact via method calls: classes `Account` and `AccountRegistry` perform sensitive operations, thus being secured in an enclave. However, class `Person` is untrusted and will not be included in the enclave.

```
1  public class Person {
2    private int hash;
3    public Person(String s, int v) {
4      byte[] buf = serialize(s);
5      CCharPointer ptr = getPointer(buf);
6      this.hash = getHash(this);
7      ocall_relayPerson(this.hash, ptr, v);
8    }
9    public void Account getAccount() {
10     ocall_relayGetAccount();
11   }
12   public void transferPerson(Person p, int v) {
13     ocall_relayTransferPerson(p.getHash(), v);
14   }
15 }
```

Listing 2: Proxy for the untrusted class `Person`.

5.1 Partitioning language

When partitioning applications destined for enclaves, an important question to answer is how to specify what should be secured or not. Some recent work proposed annotation of sensitive data [30], others proposed annotation of sensitive routines [26, 53]. While these strategies work, we argue they are not always very intuitive for an application developer. Furthermore, they require complex and expensive data-flow analysis to ensure sensitive data is not leaked.

Instead, we propose a technique based on *class annotations*. Classes are a fundamental building block for object-oriented applications, and it is very intuitive to reason about security along class boundaries. Using annotations, developers can easily specify which classes need to be secured and which ones do not. Also, annotating whole classes instead of methods or fields prevents expensive data-flow analysis to track the propagation of sensitive data within a class in order to determine other sensitive methods or fields. Hence class annotation is a more pragmatic approach.

Montsalvat supports two principal annotations: `@Trusted` and `@Untrusted`, which developers can use to specify secure and insecure classes, respectively. In Listing 1, `Account` and `AccountRegistry` classes are annotated as trusted, whereas class `Person` is untrusted.

A trusted class will always be instantiated and manipulated inside the enclave, which has two main implications. First, its member fields which are not instances of untrusted classes will be allocated on the enclave heap. Second, its methods are always executed inside the enclave.

Similarly, an untrusted class will have its instance objects allocated only on the untrusted heap, along with all its member fields which are not instances of trusted classes. All its methods will be executed outside the enclave.

Montsalvat maintains a single version of a trusted or an untrusted object in both worlds by leveraging *proxy* objects (see §5.2). In our programming model, some classes can be *neutral* as they may not be inherently trusted or untrusted. This is the case for utility classes (*i.e.*, `Arrays`, `Vector`, `String`) or other similar application-specific classes added by the developer.

Such classes are not security-sensitive and can be accessed in or out of the enclave without the use of proxies. Contrary to trusted and untrusted classes, neutral class instances can have several copies in both worlds and may evolve independently. The `@Neutral` annotation is optional, *i.e.*, classes that are not annotated are by default neutral.

One may legitimately question the relevance of two annotations, thinking the `Trusted` annotation is sufficiently expressive. Our argument for an `@Untrusted` annotation is twofold: *(1)* some classes may perform many system-related operations that are not supported inside enclaves, and keeping them in the enclave needlessly increases the TCB as they will perform many `ocall` transitions to the outside; *(2)* classes which could introduce potential security vulnerabilities in the enclave should preferably be kept out of the enclave. The `@Untrusted` annotation solves these problems, while also allowing for easy distinction with neutral classes.

Assumptions. We assume all *annotated* classes are properly encapsulated (*i.e.*, class fields are private). On the one hand, this prevents complex and expensive data flow analysis to ensure sensitive class fields do not leave the enclave. On the other hand, it guarantees that all class fields can only be accessed from outside classes via public getters and setters exposed by the class. As such, it is easier to control access to these sensitive class fields by applying techniques such as transparent encryption/decryption at the level of these public methods. Encapsulation being one pillar of object orientation, we believe this assumption to be reasonable.

5.2 Bytecode transformation

The artefacts of the partitioned application consist of two native images: a *trusted* and an *untrusted* image. The trusted image will not have any untrusted functionality, and the untrusted image will not have any trusted functionality. However, trusted objects (*i.e.*, instances of trusted classes) may call untrusted objects (*i.e.*, instances of untrusted classes) and vice versa. Hence we need to have a bidirectional communication mechanism for code flow execution. For that purpose, we introduce the notion of *proxy classes*: instances of untrusted classes have proxies in the trusted runtime, and conversely instances of trusted classes have proxies in the untrusted runtime. These proxies will serve as gateways to access the functionalities (*i.e.*, methods) of their real class in the opposite runtime.

The proxy classes expose the same methods as the original classes and replace the method implementations by a transition logic to access the original functionalities across enclave boundaries. This design makes cross-enclave object communication easier and helps maintain the object-oriented nature of the program as a whole after it is partitioned. We rely on bytecode transformations to *create* these proxy classes and *inject* code into existing classes to implement the enclave transitions. Montsalvat uses Javassist [25], a popular bytecode transformation framework, to achieve this phase.

Montsalvat automatically introduces matching proxy classes for all trusted and untrusted classes. The points-to analysis of GraalVM native-image automatically prunes/removes proxies for classes that are not reachable, which removes unnecessary proxies. As GraalVM does not include unreachable proxy classes in the generated native images (see §5.3), we did not include that analysis in the bytecode transformer. Listings 2, 3 and 4 illustrate the result of bytecode transformations for the corresponding classes.

For the purposes of the trusted image, this process creates proxy classes for untrusted classes by stripping the methods (*i.e.*, removing the method bodies) of the untrusted classes. Listing 2 shows the corresponding proxy class for the untrusted class `Person` in our illustrative example. The bodies of the stripped methods are replaced with native routines which will perform `ocall` transitions to the corresponding method in the untrusted runtime (lines 7, 10, 13 in Listing 2). Analogously, for untrusted image generation, the bytecode transformer creates proxy classes for trusted classes by stripping all methods of trusted classes, and replacing the method bodies with native methods which will perform `ecall` transitions. Listing 3 shows the corresponding proxy class for the trusted class `Account` in our illustrative example. The proxy class fields are removed and a `hash` field is added to each proxy class which stores the hash of the proxy object (*i.e.*, line 6 in Listing 3). Our present implementation uses a hash function based on Java identity hash codes. To minimize hash collisions, a hashing algorithm like MD5 [41] should be used. The result of the stripping operations is the removal of all untrusted functionality from the trusted runtime, and conversely. Only annotated classes are modified by the bytecode weaver, *i.e.*, neutral classes are not changed.

```
1  public class Account {
2    private int hash;
3    public Account(String s, int b) {
4      byte[] buf = serialize(s);
5      CCharPointer ptr = getPointer(buf);
6      this.hash = getHash(this);
7      ecall_relayAccount(this.hash, ptr, b);
8    }
9    public void updateBalance(int v) {
10     ecall_relayUpdateBalance(this.hash, v);
11   }
12 }
```

Listing 3: Proxy for the trusted class `Account`.

In the rest of the paper, we refer to all unstripped classes as *concrete classes* and stripped classes as *proxy classes*. We respectively call the instances of these classes *concrete objects* and *proxy objects*. If a concrete object in one runtime (trusted or untrusted) has a correspondence with a proxy object in the opposite runtime, we refer to that concrete object as a *mirror object* (*i.e.*, the proxy's mirror copy).

Relay methods. For the methods in one runtime (a native image) to be callable from the other runtime (another native image), these methods must be exported as *entry points*. GraalVM native-image provides an annotation (`@CEntryPoint` [37]) for specifying entry point methods which can be callable from C. These entry point methods must be static, they may only have non-object parameters and return types, *i.e.*, primitive types or `Word` types (including pointers) [35], and they must specify the GraalVM isolate that will serve as execution context for the method. As a result of these restrictions, it is not feasible to export all methods of concrete classes as entry points directly, as this would require changing their signatures. To circumvent this limitation, we introduce static entry point methods to act as wrappers for the invocations of the associated class or instance methods. We call these *relay methods*.

For every public method in a concrete class, including all constructors, the bytecode transformer adds an associated relay method to the class. The native methods (*i.e.*, `ecall` and `ocall` routines) we added to the stripped methods of the proxy classes will perform enclave transitions to invoke the corresponding relay methods.

The parameters of a relay method comprise: an isolate which provides the execution context for the method call, the hash of the calling proxy object (for non-static methods), all primitive parameters of the associated method, and pointers (*i.e.*, `CCharPointer` [36])

```
1  public class Account {
2    private String owner;
3    private int balance;
4    public Account(String s, int b) {...}
5    public void updateBalance(int v) {...}

7    @CEntryPoint
8    public static void relayAccount(Isolate ctx, int hash, CCharPointer buf, int b) {
9      String s = deserialize(buf);
10     Account mirror = new Account(s, b);
11     mirrorProxyRegistry.add(hash, mirror);
12   }
13   @CEntryPoint
14   public static void relayUpdateBalance(Isolate ctx, int hash, int v) {
15     Account mirror = mirrorProxyRegistry.get(hash);
16     mirror.updateBalance(v);
17   }
18 }
```

Listing 4: Concrete class `Account`, once transformed.

```
1  public void addAccount(Account acc) {
2    ecall_relayAddAccount(acc.getHash());
3  }

5  @CEntryPoint
6  public static void relayAddAccount(Isolate ctx, int hash) {
7    Account mirror = mirrorProxyRegistry.get(hash);
8    this.addAccount(mirror);
9  }
```

Listing 5: Proxy (top) and relay (bottom) methods for the `AccountRegistry` class.

which represent the addresses of buffers obtained from the serialization of any object parameters which are instances of neutral classes (as these classes do not need proxies). For proxy object parameters, the hash of the corresponding proxy is passed as parameter and the corresponding mirror object will be used as the parameter once the real (*i.e.*, concrete) method is called in the opposite runtime. Similarly, for mirror object parameters, the hash of the corresponding proxy is also sent, and the corresponding proxy object is used in the opposite runtime as parameter in the method.

As for the serialized buffers, they are deserialized and the corresponding object parameter recreated in the body of the relay method. For relay methods of constructors, we add code to instantiate the corresponding mirror object, as well as code to add the mirror object strong reference and associated proxy hash to a global registry, which we call the *mirror-proxy registry*. For instance methods, we add code to look up the corresponding mirror object in the registry, and then invoke the instance method on that mirror object with its corresponding parameters. Neutral object return types from the untrusted runtime are also serialized and copied across the enclave boundary. Both the trusted and untrusted runtimes have a mirror-proxy registry.

The code in Listing 4 outlines the state of concrete class `Account` after bytecode transformation. For illustration, the relay method `relayAccount` (line 8) is added into concrete class `Account` in the trusted runtime automatically.

We complete the transformation of the other classes of our illustrative example as follows. For proxy class `Person`, we will have `ocalls` instead to the relay methods. For proxy class `AccountRegistry`, we have a proxy `Account` as parameter in the `addAccount` method, and only its hash will be passed to the opposite runtime. The resulting proxy method and the corresponding relay method in concrete class `AccountRegistry` are shown in Listing 5.

Native transition methods. The native transition methods (*e.g.*, `ecall_addAccount`) are C routines which perform enclave transitions to the opposite runtime. At the time of bytecode transformation, the definitions of the native transition methods are absent and only their signatures are provided. Their definitions will be generated by the native image generator (see the next section).

5.3 Native image partitioning

The GraalVM native image generator is responsible for building native images. It takes as input compiled application classes (bytecode) and all their associated external libraries, including the JDK. The native image generator then performs points-to analysis [57] to find the reachable program element (classes, methods and fields). Only reachable methods are then compiled ahead-of-time into the final native image. For an SGX-based environment, this let us exclude any redundant application logic from the enclave. The resulting image embeds runtime components for garbage collection (memory management), thread scheduling, *etc.*

Static analysis for trusted and untrusted image generation. The bytecode transformations produced two sets of class files: the first set (T) comprises modified trusted classes and untrusted proxy classes, while the second set (U) comprises modified untrusted classes and secure proxy classes. The unannotated/neutral classes (N) were not changed by the bytecode transformer. These three sets of classes are used by GraalVM to generate two native images, *i.e.*, the partitioned application.

The native image generator in Montsalvat uses set ($T \cup N$) as input for *trusted image* generation and the set ($U \cup N$) as input for *untrusted image* generation. To determine reachable program elements (*i.e.*, classes, fields and methods) the native image generator leverages a static analysis technique known as points-to analysis [48, 57]. Points-to analysis starts with all entry points and iteratively processes all transitively reachable classes, fields and methods [57]. For the sake of brevity, we do not include all the steps performed during points-to analysis in GraalVM native-image (see [57] for details). For the trusted image, all the relay methods of trusted classes will serve as entry points (recall the `@CEntryPoint` annotation, *e.g.*, in Listing 4). For the untrusted image, the main entry point (Java application's `main` method) and the relay methods of untrusted classes will serve as entry points. Conceptually, we can include the main entry point in either the trusted or untrusted image. However we chose to add it in the untrusted image because: *(1)* it prevents an `ecall` transition to invoke the main method and `ocall` transition to create garbage collection helper threads (see §5.5) once in the main method, and *(2)* it is in accordance with Intel SGX's programming convention, as all SGX applications begin in the untrusted runtime.

Figure 2 illustrates a simplified reachability analysis done for two entry point methods (`relayAccount` and `main`) to determine their reachable methods. A similar process is performed for all other entry points.

Once static analysis is complete, the trusted image no longer contains untrusted methods/functionality. It embeds proxies instead, in case some untrusted proxy class methods were reachable. Similarly, the untrusted image does not contain trusted methods, but only proxies to those if some proxy class methods were reachable. Following from our illustrative example, proxy class `Person` will

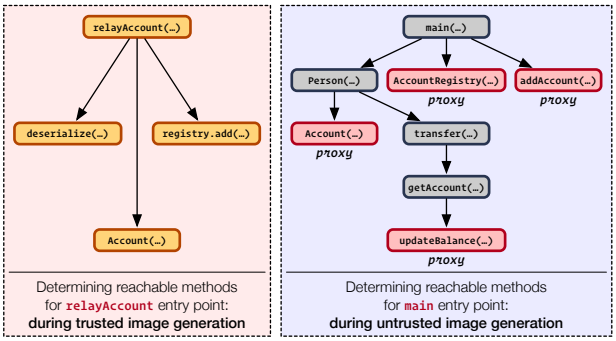

Figure 2: For the trusted image, the relay methods in the trusted classes ensure that other trusted class methods, as well as methods from neutral classes (*e.g.*, `serialize`, `registry.add`, *etc.*), are reachable. Similarly, for the untrusted image, the main entry point ensures that the `Person` class methods, as well as methods from proxy classes (`Account` and `AccountRegistry`), are reachable. This is a subset of the full graph (other methods will be made reachable at leaf nodes).

```
1 void ecall_relayAddAccount(int hash) {
2   Isolate ctx = getIsolate();                    // Get the enclave isolate
3   relayAddAccount(ctx, hash);
4 }
```

Listing 6: C code for `ecall_relayAddAccount`.

not be included inside the trusted image since it is not reachable from any of the trusted classes.

In the `main` method in our illustrative example, at runtime during object creation, the constructors of `Person` $p1$ and $p2$ will instantiate a proxy object of the *proxy* `Account` class. The string parameters `"Alice"` and `"Bob"` will be serialized and an `ecall` transition will be made to create a corresponding *mirror* `Account` object in the enclave. Similarly, when `p1.transfer(p2, 25)` is called, the corresponding proxy `Account` objects will perform enclave transitions to update the balances in the enclave. In the same way, the call to the proxy `AccountRegistry` constructor performs a transition to create a mirror object in the enclave corresponding to proxy object `reg`. The latter performs a transition too when `addAccount` is called.

By default, the native image generator compiles all reachable methods and links the latter with GraalVM's native libraries to produce an executable or a shared object file. We modified the image generator to bypass the linking phase that produces executables or shared objects, so as to produce relocatable object files (`.o`) which can be linked to other libraries to build the final SGX application. The resulting images for the trusted and untrusted parts are `trusted.o` and `untrusted.o` respectively. These will be dispatched to the SGX module and used to build the final SGX application.

SGX code generator. During bytecode transformation, native `ecall` and `ocall` transition routines are added to proxy classes. We extended GraalVM native-image with a class to generate C code definitions for the corresponding `ecall` (added in trusted proxy classes) or `ocall` (added in untrusted proxy classes) transitions, as well as their corresponding header files. Listing 6 shows the generated C code for the `ecall_relayAddAccount` method.

The code generator also creates associated EDL files used by the *Edger8r* tool in the Intel SGX SDK to build bridge routines, which marshal the data across the enclave boundary. The generated files are dispatched to the SGX module.

5.4 SGX application creation

This is the final stage in the Montsalvat workflow. Because SGX enclaves operate only in user mode, they cannot issue system calls and standard OS abstractions (*e.g.*, file systems, network), which are ubiquitous in real-world applications. The solution to this problem is to relay these unsupported calls to the untrusted runtime, which does not have the same limitations. In contrast to systems that introduce a *LibOS* (*i.e.*, an entire operating system implemented as a library) in the enclave, we leverage an approach which involves redefining unsupported `libc` routines as wrappers for `ocalls`. These redefined `libc` routines in the enclave constitute Montsalvat's *shim library*. The latter intercepts calls to unsupported `libc` routines and relays them to the untrusted runtime. A *shim helper library* in the untrusted runtime then invokes the real `libc` routines. This by design reduces the TCB when compared to *LibOS*-based systems.

Montsalvat then compiles all generated `ecall` routines and statically links them with the trusted image (`trusted.o`), the shim library and native libraries from GraalVM to produce the final enclave shared library, which corresponds to the trusted part of the Java SGX application. Similarly, the generated `ocall` routines are also compiled and linked with the untrusted image (`untrusted.o`) and GraalVM native libraries to produce the final untrusted component. In accordance with Intel SGX's application model, Montsalvat compiles the main entry point of partitioned applications in the untrusted component. The resulting trusted and untrusted components compose the final SGX application. At runtime, a GraalVM isolate is created in both the trusted and untrusted part of the application. These isolates provide the execution contexts for transition routines, *i.e.*, the trusted isolate serves `ecall` routines while the untrusted isolate serves `ocall` routines.

5.5 Garbage collection

Following our partitioned application design, untrusted code objects can have trusted counterparts (proxies) and vice versa. This presents a challenge at the level of garbage collection (GC) because we must ensure synchronised destruction of objects across the trusted and untrusted heaps. More specifically, we need to synchronise GC of proxy and mirror objects, *e.g.*, the mirror of proxy object `reg` should not be destroyed before `reg` in our illustrative example (Listing 1). Similarly, when `reg` is destroyed, its corresponding mirror object should be made eligible for GC.

Java provides *finalizer methods* [38] which the garbage collector invokes prior to garbage collecting an object. So one could envision a solution based on finalizer methods. However, the latter are deprecated since Java 9 and have badly designed semantics [34]. For example, a finalizer method can make a proxy object reachable again, which will lead to an inconsistent state across the trusted and untrusted heaps after the proxy's mirror object is destroyed.

To address this problem, we implemented an application-level GC helper based on `weak references`. When a proxy object is created, Montsalvat stores a weak reference and the hash of the former in a global list. We use a weak reference here because it does not prevent the proxy object from being garbage collected once it is eligible for GC. The GC helper thread periodically (*e.g.*, every second) scans this list for null referents of weak references,

i.e., objects referred to by the weak references. Once such a null referent is found, it means the proxy object has been (or is eligible for being) garbage collected, and thus we can remove the corresponding mirror object from the mirror-proxy registry in the opposite runtime. This makes the mirror object eligible for GC if it is not strongly referenced anywhere else. Both the trusted and untrusted runtimes maintain independent lists of proxy weak references for the associated runtime, and two GC helper threads are spawned in the application: one to scan the trusted list in the enclave, and the other to scan the untrusted list.

5.6 Running unpartitioned native images

Despite the benefits of partitioning an enclave application, situations may arise where it is much easier for the application developer to run the entire application as a native image inside the enclave. This could happen when the majority of classes potentially deal with sensitive information and no classes qualify as untrusted. Consequently, Montsalvat makes it possible to run unpartitioned applications. Unpartitioned applications do not require annotations, hence no bytecode modifications are performed. The original application is built into a single native image which is linked entirely to the final enclave object.

5.7 Prototype implementation

Our current Montsalvat prototype is based on GraalVM CE v21.0.0 for Java 8. GraalVM and the bytecode transformer are implemented in Java. Our modifications in GraalVM amount to ~1,400 lines of code (LOC). The bytecode transformer relies on Javassist v3.26 and contains ~2,100 LOC. The SGX module is based on SGX SDK and SGX driver v2.11. It consists of ~10,200 C/C++ LOC. We plan to release Montsalvat as open-source.

6 EVALUATION

This section presents an experimental evaluation of Montsalvat based on micro- and macro-benchmarks with real-world applications. We seek to answer the following questions:

- Q_1) What is the cost of proxy object creation and remote method invocations? (§6.2, §6.3)
- Q_2) How does partitioning impact GC performance? (§6.4)
- Q_3) How does partitioning impact application performance? (§6.5)
- Q_4) How do partitioned and unpartitioned native images in SGX enclaves compare with applications running on a JVM in enclaves? (§6.6)

6.1 Experimental setup

Our evaluation is conducted on a server equipped with a quad-core Intel Xeon E3-1270 CPU clocked at 3.80 GHz, and 64 GB of DRAM. The processor has 32 KB L1i and L1d caches, 256 KB L2 cache and 8 MB L3 cache. The server runs Ubuntu 18.04.1 LTS 64 bit and Linux kernel 4.15.0-142. We run the Intel SGX platform software, SDK and driver version v2.11. The EPC size on this server is 128 MB, of which 93.5 MB is usable by enclaves. The enclaves have maximum heap sizes of 4 GB and stack sizes of 8 MB. All native images are built with a maximum heap size of 2 GB.

We use SCONE to run unmodified applications on a JVM in SGX enclaves. The SCONE containers are based on Alpine Linux [1]. The base SCONE image ships OpenJDK8 (tag: `8u181-jdk-alpine-scone5.1.0`). For non-SCONE experiments, we execute the GraalVM compiler, which generates the native images, in OpenJDK-8u282. At execution, only the code of the generated native images runs. All reported latencies are averaged over 5 runs.

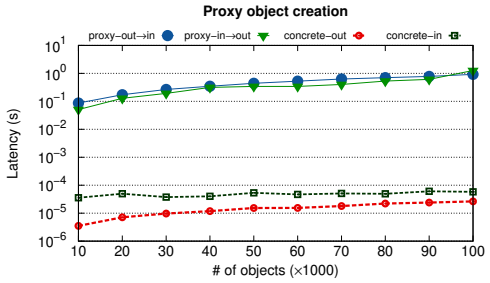

Figure 3: Performance of proxy object creation vs. concrete object creation.

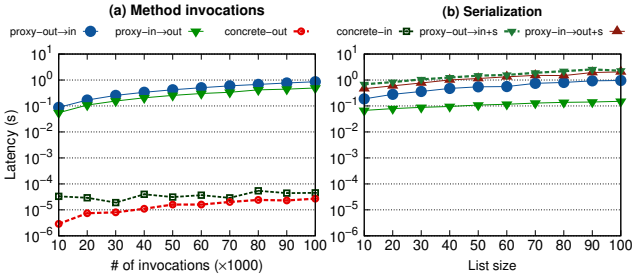

Figure 4: Performance of remote method invocations (RMIs) by proxy objects vs. concrete object invocations, and impact of serialization on RMIs.

6.2 Performance of proxy creation

(Answer to Q_1) The goal of this experiment is to study the latency of proxy creation in relation to normal (concrete) object creation. The notion of *proxy* is an internal feature of Montsalvat which could impact application performance. We use a synthetic Java program to realise this experiment. Figure 3 shows the results obtained. We perform object instantiations in four different scenarios: concrete object creation in and out of the enclave methods (respectively labelled `concrete-in` and `concrete-out` in Figure 3), and proxy object in (`proxy-in`→`out`) and out (`proxy-out`→`in`) of the enclave. Scenario `concrete-out` corresponds to the base line for this experiment.

We observe that proxy object creation latency in the enclave is 3 orders of magnitude higher when compared to concrete object creation in the enclave, and proxy object creation latency out of the enclave is 4 orders of magnitude higher when compared to concrete object creation out of the enclave. This performance drop when creating proxy objects is mainly due to the expensive enclave transitions required to instantiate the corresponding mirror objects in the opposite runtime.

6.3 Performance of RMI and impact of serialization

(Answer to Q_1) The goal of this experiment is to study the performance of remote method invocations by proxy objects, and understand the impact of serialization on these invocations. To this end,

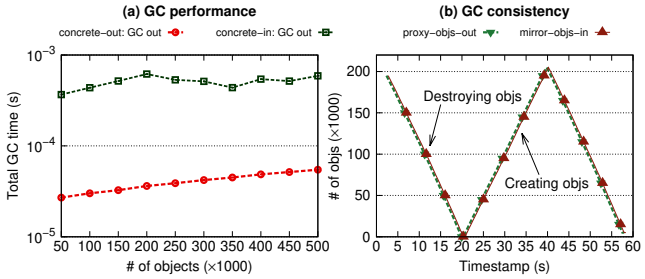

Figure 5: Garbage collection performance.

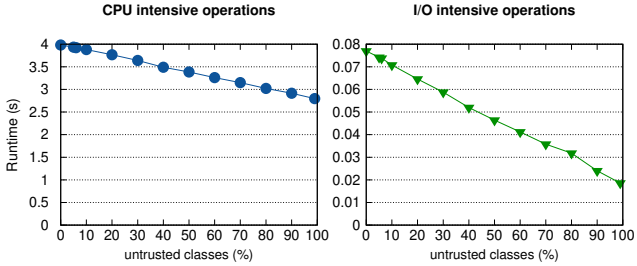

Figure 6: Enclave performance is better when fewer classes are in the enclave.

we generate synthetic programs where objects perform method invocations in four different scenarios: concrete object invoking instance methods in and out of the enclave (respectively labelled `concrete-in` and `concrete-out` in Figure 4 (a)) and proxy object invoking instance methods remotely from within (`proxy-in`⟶`out`) and out of (`proxy-out`⟶`in`) the enclave without serializable parameters. We vary the number of method invocations of the objects in these scenarios and calculate the corresponding latency of the invocations.

Figure 4 (a) shows that when there is no serialization involved, the latency of proxy object RMI in the enclave is 3 orders of magnitude higher than the latency of concrete object method invocation in the enclave. On the other hand, the latency of proxy object RMI out of the enclave is 4 orders of magnitude higher when compared to concrete object method invocation latency. This overhead is similar to that observed in proxy object creation, and is mainly due to the expensive enclave transitions involved.

To understand the impact of serialization on proxy method invocations, we introduce two more scenarios where proxies in and out of the enclave invoke methods with a serializable parameter (respectively labelled `proxy-in`⟶`out+s` and `proxy-out`⟶`in+s` in Figure 4). The serialized parameter is a list of 16 byte string values. We vary the size of the serialized list while keeping the number of method invocations constant at 10,000 invocations. For scenarios `proxy-in`⟶`out` and `proxy-out`⟶`in`, we use the same methods as in the `...+s` variants but without passing the list as parameter.

Figure 4 (b) shows that RMIs in the enclave with the serialized parameter are about 10× more expensive than the corresponding RMIs without serialization, while RMIs out of the enclave are about 3× more expensive than the corresponding RMIs without serialization.

It should be noted that the orders of magnitude and ratios calculated will vary depending on the latency of the method operations themselves, without taking into account the cost of parameter serializations or enclave transitions. The methods used in these experiments are setter methods updating an object field, which are relatively inexpensive operations. For more expensive methods, the cost of the method operations should outweigh the cost of enclave transitions or parameter serializations, hence decreasing the importance of the latter.

6.4 Garbage collection performance

(Answer to Q_2) To understand the performance variations of garbage collection in and out of the enclave, we performed an experiment which involves creating multiple concrete objects, making them eligible for GC and invoking the garbage collector in and out of the enclave. We record the total time spent for garbage collection in both scenarios. Figure 5 (a) shows the results obtained. We observe that the enclave adds an order of magnitude more overhead to the garbage collection operation. GraalVM native images embed a serial stop and copy GC [35]; the copy operation of this GC in the enclave leads to more data exchange between the CPU and the EPC, hence the overhead when compared to GC performance outside (*i.e.*, `concrete-out`).

We performed a second experiment to demonstrate garbage collection consistency in and out of the enclave. In this experiment a synthetic Java program creates proxy objects in the untrusted runtime, makes some of the objects eligible for GC and invokes the GC in the untrusted runtime. This operation is repeated for a given time range. The number of live (not garbage collected) proxy objects out of the enclave and the number of mirror objects in the enclave mirror-proxy registry are recorded at different timestamps. Figure 5 (b) shows the results obtained. We observe that as proxy objects are garbage collected, mirror objects are removed from the in-enclave mirror-proxy registry, making them eligible for GC too. In the same way, as more proxies are created, we notice a similar increase in the number of mirror objects in the enclave. These results show that GC is consistent between the trusted and untrusted image.

6.5 Speed up due to partitioning

(Answer to Q_3) To demonstrate the performance improvements of partitioning native applications for enclaves with Montsalvat, we leverage a synthetic Java program, and two real-world applications, *i.e.*, PalDB [31] and GraphChi [29].

Synthetic benchmark. We developed a Java program generator to create Java applications with various numbers of classes annotated as trusted or untrusted. We generated a Java application with 100 classes. Each class contains an instance method which performs either CPU intensive operations (*i.e.*, compute a fast Fourier transform [14] on a 1 MB double array) or I/O intensive operations (*i.e.*, writes 4 KB of data to a file). The main method instantiates each class and invokes the associated instance method. We vary the number of trusted and untrusted classes for two scenarios: *(1)* all class instance methods perform CPU intensive operations and *(2)* all class instance methods perform I/O intensive operations. We then calculate the total execution time of the resulting application. Figure 6 shows the results.

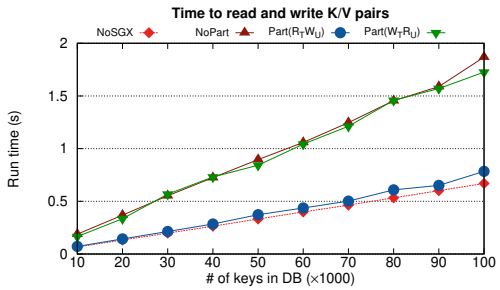

Figure 7: Read and write times for partitioned PalDB.

We observe that, as the percentage of untrusted classes increases (*i.e.*, more classes are moved out of the enclave), the overall application runtime improves. For I/O operations, fewer enclave transitions are done for I/O write operations, leading to better performance. For CPU operations, enclave performance can get more expensive when random reads and writes are done on data which is not present in the CPU [4, 56]. This decrease in performance is caused by on-the-fly encryption/decryption of CPU cache-lines by the MEE [56] when data is transferred between the CPU and the EPC. In summary, this synthetic benchmark suggests that by delegating computations to the untrusted runtime, we relieve the enclave of expensive computations, leading to better enclave performance, and better overall application performance. We illustrate this further with the two real world applications below.

PalDB. PalDB is an embedabble persistent key-value store developed by LinkedIn, used in analytics workflows and machine-learning applications. We consider a Java application based on PalDB which writes and reads a list of key-value (K/V) pairs in a store file. The keys are string values of randomly generated integers (in the range $[0, 2^{31} - 1]$), while the values are randomly generated strings of length 128. For this, we introduced two classes: `DBReader` and `DBWriter` which exploit PalDB's API for respectively reading from and writing to the store file. A natural and intuitive partitioning scheme for this application is to partition along the `DBWriter` and `DBReader` classes, depending on the security requirements of the application. For this we consider two possible scenarios: `DBReader` trusted and `DBWriter` untrusted ($R_T W_U$), and `DBReader` untrusted and `DBWriter` trusted ($R_U W_T$). We run the unpartitioned application (base line) as a native image in an SGX enclave and compare its performance to the partitioned version with the above mentioned schemes, as well as the native image running without SGX enabled. Figure 7 shows the results obtained.

For both partitioning schemes ($R_T W_U$ and $R_U W_T$) we observe performance improvements after partitioning the application. $R_T W_U$ is on average 2.5× times faster while $R_U W_T$ is on average 1.04× faster when compared to the unpartitioned native image. PalDB optimises reads by memory mapping the store file in memory, but does regular I/O for writes to the store file. This explains the greater performance improvement after partitioning using $R_T W_U$ as it relieves the enclave of expensive write-induced enclave transitions to perform I/O, making it closer to the native performance (no SGX). For $R_U W_T$ the performance improvement is less as more `ocall` transitions (23× more on average than $R_T W_U$) are done to write K/V pairs to the store file from within the enclave.As expected,

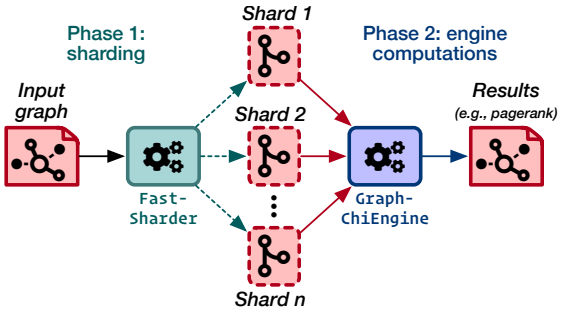

Figure 8: Typical GraphChi program workflow.

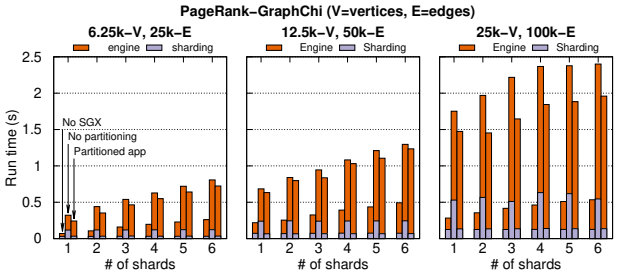

Figure 9: Execution time for partitioned PageRank.

the application has best performance when running without SGX, however this is the most insecure configuration.

GraphChi. GraphChi is a large-scale graph processing framework. We use the popular *PageRank* [2] algorithm as an example application for partitioning. PageRank evaluates the relative importance of nodes in a directed graph. Typically, GraphChi applications follow the workflow outlined in Figure 8. The input graph is split by a sharder (`FastSharder`) into multiple parts (shards) which are then processed in the core execution engine (`GraphChiEngine`) to produce the final result (PageRank values in our case). A possible partitioning scheme for the application would be along the `FastSharder` and `GraphChiEngine` classes. For this we make the `GraphChiEngine` trusted and the `FastSharder` untrusted. We run the PageRank algorithm on synthetic directed graphs generated using the RMAT algorithm [11]. We vary graph sizes by varying the number of vertices (V) and edges (E) in the graph. For each graph, we vary the number of shards for the PageRank computations and compare the performance of the partitioned native image to the unpartitioned case, as well as the native image running without SGX. Figure 9 shows the results obtained.

For each shard, the leftmost bar shows the run time without SGX, the middle bar for the unpartitioned native image running inside the enclave, and the rightmost bar for the partitioned application. We show the total times to calculate the PageRank values of the graph nodes, as well as the portion of the total time spent in sharding and in the engine.

After partitioning, the `FastSharder` is transferred to the untrusted runtime, relieving the enclave of all expensive I/O related work done by the sharder, thereby improving enclave latency. We can observe on the graph that the latency due to sharding after partitioning is approximately the same as the native case (no SGX),

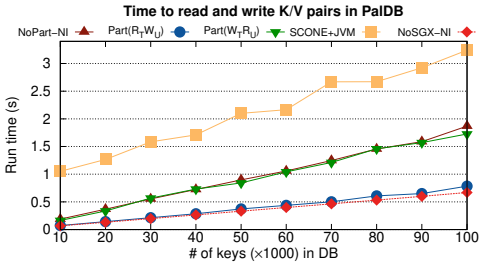

Figure 10: Partitioned and unpartitioned PalDB native images vs. PalDB in SCONE+JVM.

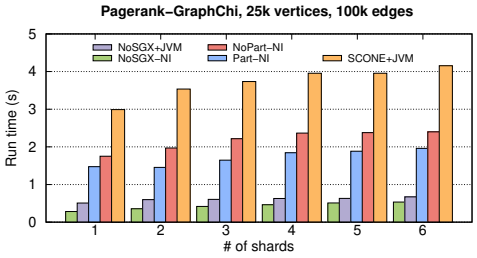

Figure 11: Partitioned and unpartitioned GraphChi native images vs. GraphChi in SCONE+JVM.

which is explained by the fact that the `FastSharder` now operates in the untrusted runtime and there is no extra overhead due to MEE encryption/decryption operations in the enclave. This leads to a performance gain of about 1.2× on average as compared to the unpartitioned case. We observe similar performance improvements for the different graph sizes.

6.6 Comparing JVM-based applications in enclaves

(Answer to Q_3) To understand how partitioned and unpartitioned native images compare to the JVM-based counterparts running in enclaves, we compared SGX-based native images to the applications running on a JVM in a SCONE container. The JVM is run with maximum heap size of 2 GB (`-Xmx2G`). We are not particularly concerned with the performance of applications running on a JVM out of enclaves. However we included results for the latter to get a clearer picture of the performance variations using the different approaches.

Partitioned native images vs. JVM-based applications in enclaves. For this experiment we compared the partitioned versions of PalDB and GraphChi, using the same partitioning schemes, to their unpartitioned counterparts running on a JVM in a SCONE container. Figures 10 and 11 show the results for PalDB and GraphChi respectively.

From the results, we observe that $R_T W_U$ and $R_U W_T$ are respectively 6.6× and 2.8× faster on average when compared to PalDB running on a JVM in SCONE. As for GraphChi, the partitioned GraphChi native image is 2.2× faster on average when compared to GraphChi running on a JVM in SCONE. The poor performance of the applications with the JVM in SCONE can be justified by two reasons: *(1)* the JVM spends some time for class loading, bytecode interpretation and dynamic compilation; these operations are absent in native images, *(2)* the in-enclave JVM increases the number objects in the enclave heap, which leads to more data exchange between the EPC and CPU, hence more expensive MEE encryption/decryption of CPU cache lines when compared to the native images in the enclaves.

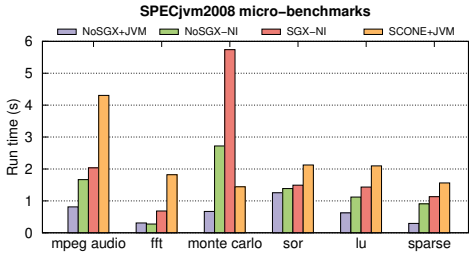

Figure 12: Performance of unpartitioned SPECjvm 2008 micro-benchmarks in enclaves.

Benchmark name	Latency gain over SCONE+JVM
Mpegaudio	2.12×
FFT	2.66×
Monte_Carlo	0.25×
SOR	1.42×
LU	1.46×
Sparse	1.38×

Table 1: Ratio between unpartitioned SPECjvm2008 native images in enclaves (`SGX-NI` in Figure 12) against their on-JVM counterparts in SCONE (`SCONE+JVM`).

Unpartitioned native images in enclaves vs. JVM-based applications in enclaves. Here we compare the performance of unpartitioned native images in enclaves to their JVM-based counterparts. We present results for PalDB (Figure 10) and GraphChi (Figure 11), as well as 6 SPECjvm2008 [49] micro-benchmarks (Figure 12) using their default workloads.

The unpartitioned PalDB and GraphChi native images in the enclave are respectively 2.6× and 1.7× faster when compared to their JVM counterparts in a SCONE container. For the SPECjvm2008 micro-benchmarks, Table 1 summarises the comparisons of the native images vs. their JVM counterparts, all running within enclaves. We observe comparatively lower performance for the JVM counterparts in a SCONE container except for the *Monte_Carlo* micro-benchmark. We explain this to garbage collection cycles triggered in the native image. Recent studies [28] suggest the GC in GraalVM native-image performs poorly when compared to OpenJDK HotSpot JVM's garbage collectors. The poorer in-enclave JVM results for the rest can be justified by the two reasons mentioned previously.

6.7 Additional use-case scenarios

Montsalvat can be used for a wide variety of security applications. Examples include secure key/value stores and blockchain applications. The classes/business logic for storing and retrieving key/value pairs, and business logic for smart contracts can be secured in the enclave, while classes for network-related functionality are kept

out of the enclave. The partitioned components then interact in accordance with our design.

7 CONCLUSION AND FUTURE WORK

This paper presented Montsalvat, a tool for automatically partitioning Java applications destined for secure enclave environments. Montsalvat leverages source code annotations and bytecode transformations to partition application classes into trusted and untrusted versions. Montsalvat provides an RMI-like mechanism to enable object communication between the partitioned components, as well as a garbage collection extension to ensure consistent garbage collection across the trusted and untrusted application heaps. We implemented Montsalvat atop GraalVM native-image, and our extensive evaluations show Montsalvat can provide strong security guarantees while improving application performance as compared to systems that execute complete applications together with the associated runtime in an enclave. We intend to extend this work along the following directions. First, we will improve Montsalvat's RMI system with transition-less cross-enclave calls for expensive RMIs, similar to [51], especially useful for applications performing several enclave transitions. Second, we plan to extend our proxy-mirror system to permit creation and interaction of proxy-mirror object pairs across multiple isolates in both the trusted and untrusted runtimes.

ACKNOWLEDGMENTS

This work is supported in part by Oracle donations CR 2901 and CR 3801, as well as project 200021_178822 of the Swiss National Science Foundation (FNS). We would also like to thank the anonymous reviewers and our shepherd, Roy H. Campbell, for their feedback.

REFERENCES

[1] Alpine Linux. https://alpinelinux.org/.
[2] Alon Altman and Moshe Tennenholtz. Ranking systems: the PageRank axioms. In *Proceedings of the 6th ACM conference on Electronic Commerce (EC 05)*, Vancouver, BC, Canada, 2005.
[3] Tiago Alves. Trustzone: Integrated hardware and software security. *White paper*, 2004.
[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, USA, 2016.
[5] Mike Barnett, Manuel Fahndrich, Francesco Logozzo, and Diego Garbervetsky. Annotations for (more) Precise Points-to Analysis. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO 2007)*, Berlin, Germany, 2007.
[6] Paulo Barros, Rene Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 669–679, 2015.
[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–26, 2015.
[8] Daniele Bonetta. GraalVM: metaprogramming inside a polyglot system (invited talk). In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection (SPLASH 18)*, Boston, MA, USA, 2018.
[9] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper using intel SGX. In *Proceedings of the 17th International Middleware Conference (Middleware 16)*, New York, NY, USA, 2016.
[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, USA, 2018.
[11] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM 04)*, Lake Buena Vista, FL, USA, 2004.
[12] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. OPERA: Open remote attestation for Intel's secure enclaves. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 19)*, London, United Kingdom, 2019.
[13] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. VoltPillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *30th USENIX Security Symposium (USENIX Security 21)*, Online, 2021.
[14] W.T. Cochran, J.W. Cooley, D.L. Favin, H.D. Helms, R.A. Kaenel, W.W. Lang, G.C. Maling, D.E. Nelson, C.M. Rader, and P.D. Welch. What is the fast fourier transform? *Proceedings of the IEEE*, 55(10):1664–1674, 1967.
[15] Intel Corporation. SDK for Intel© Software Guard Extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/sdk.html.
[16] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016(86):1–186, 2016.
[17] Docker, inc. Docker: Empowering App Development for Developers. https://www.docker.com.
[18] Pratik Fegade and Christian Wimmer. Scalable pointer analysis of data structures using semantic models. In *Proceedings of the 29th International Conference on Compiler Construction (CC 2020)*, San Diego, CA, USA, 2020.
[19] Apache Software Foundation. Apache Hadoop. https://hadoop.apache.org/.
[20] Apache Software Foundation. Apache Spark – Unified Analytics Engine for Big Data. https://spark.apache.org/.
[21] Apache Software Foundation. Apache ZooKeeper. https://zookeeper.apache.org/.
[22] William Grosso. *Java RMI*. O'Reilly Media, 2002.
[23] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium, (USENIX Security 2017)*, Vancouver, BC, Canada, 2017.
[24] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009.
[25] Jboss-Javassist. Javassist. https://www.javassist.org/.
[26] Jianyu Jiang, Xusheng Chen, TszOn Li, Cheng Wang, Tianxiang Shen, Shixiong Zhao, Heming Cui, Cho-Li Wang, and Fengwei Zhang. Uranus: Simple, efficient SGX programming and its applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIACCS 2020)*, Taipei, Taiwan, 2020.
[27] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.
[28] Jonatan Kazmierczak. High performance at low cost – choose the best JVM and the best Garbage Collector for your needs. https://bit.ly/3f6mLjO.
[29] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Boston, MA, USA, 2012.
[30] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, USA, 2017.
[31] LinkedIn. PalDB: an embeddable write-once key-value store written in Java. https://github.com/linkedin/PalDB.
[32] A. Mogage, R. Pires, V. Craciun, E. Onica, and P. Felber. Supply chain malware targets sgx: Take care of what you sign. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 52–528, Los Alamitos, CA, USA, oct 2019. IEEE Computer Society.
[33] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 2018)*, Boston, MA, USA, 2018.
[34] Oracle. Finalizers. https://www.graalvm.org/reference-manual/native-image/Limitations/#finalizers.
[35] Oracle. GraalVM Native Image. https://www.graalvm.org/reference-manual/native-image/.
[36] Oracle. GraalVM SDK Java API Reference – CCharPointer. https://www.graalvm.org/sdk/javadoc/index.html?org/graalvm/nativeimage/c/type/CCharPointer.html.
[37] Oracle. GraalVM SDK Java API Reference – CEntryPoint. https://www.graalvm.org/sdk/javadoc/org/graalvm/nativeimage/c/function/CEntryPoint.html.
[38] Oracle. The Java Language Specification. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.
[39] Oracle. Tracing agent. https://docs.oracle.com/en/graalvm/enterprise/19/guide/reference/native-image/tracing-agent.html.

[40] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[41] Ronald Rivest and S Dusse. The md5 message-digest algorithm.

[42] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. *SIGPLAN Not.*, 36(11):43–55, October 2001.

[43] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[44] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy (SSP 15)*, San Jose, CA, USA, 2015.

[45] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 17)*, Bonn, Germany, 2017.

[46] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *24th Annual Network and Distributed System Security Symposium (NDSS 17)*, San Diego, CA, USA, 2017.

[47] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB linux applications with SGX enclaves. In *24th Annual Network and Distributed System Security Symposium (NDSS 17)*, San Diego, CA, USA, 2017.

[48] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015.

[49] SPEC. SPECjvm2008. https://www.spec.org/jvm2008/.

[50] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 16)*, ASPLOS '18, page 665–678, Williamsburg, VA, USA, 2018.

[51] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless calls made practical in intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX 18)*, New York, NY, USA, 2018.

[52] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, Santa Clara, CA, USA, 2017.

[53] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. Civet: An efficient Java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, Online, 2020.

[54] Yongzhi Wang, Yulong Shen, Cuicui Su, Ke Cheng, Yibo Yang, Anter Faree, and Yao Liu. CFHider: Control flow obfuscation with Intel SGX. In *IEEE Conference on Computer Communications (INFOCOM 2019)*, Paris, France, 2019.

[55] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. sgx-perf: A performance analysis tool for intel SGX enclaves. In Paulo Ferreira and Liuba Shrira, editors, *Proceedings of the 19th International Middleware Conference (Middleware 2018)*, Rennes, France, 2018.

[56] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. *SIGARCH Comput. Archit. News*, 45(2):81–93, June 2017.

[57] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. In *2019 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 19)*. Athens, Greece, 2019.

[58] Zhenyu Xu, Thomas Mauldin, Qing Yang, and Tao Wei. Runtime detection of probing/tampering on interconnecting buses. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 247–251, 2021.

[59] P. Yuhala, P. Felber, V. Schiavoni, and A. Tchana. Plinius: Secure and persistent machine learning model training. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 52–62, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society.

[60] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA, USA, 2017.