Lejacon: A Lightweight and Efficient Approach to Java Confidential Computing on SGX

[1]Xinyuan Miao, [2]Ziyi Lin, [2]Shaojun Wang, [2]Lei Yu, [2]Sanhong Li,
[1]Zihan Wang, [1]Pengbo Nie, [1]Yuting Chen, [1]Beijun Shen, [3]He Jiang
[1]*Shanghai Jiao Tong University*, Shanghai, China
[2]*Alibaba Group*, Shanghai, China
[3]*Dalian University of Technology*, Dalian, China
{mxinyuan, wangzh99, yuemonangong, chenyt, bjshen}@sjtu.edu.cn,
{cengfeng.lzy, jeffery.wsj, lei.yul, sanhong.lsh}@alibaba-inc.com, hejiang@dlut.edu.cn

***Abstract*—Intel's SGX is a confidential computing technique. It allows key functionalities of C/C++/native applications to be confidentially executed in hardware enclaves. However, numerous cloud applications are written in Java. For supporting their confidential computing, state-of-the-art approaches deploy Java Virtual Machines (JVMs) in enclaves and perform confidential computing on JVMs. Meanwhile, these *JVM-in-enclave* solutions still suffer from serious limitations, such as heavy overheads of running JVMs in enclaves, large attack surfaces, and deep computation stacks. To mitigate the above limitations, we formalize a *Secure Closed-World* (SCW) principle and then propose *Lejacon*, a lightweight and efficient approach to Java confidential computing. The key idea is, given a Java application, to (1) separately compile its confidential computing tasks into a bundle of *Native Confidential Computing* (NCC) services; (2) run the NCC services in enclaves on the Trusted Execution Environment (TEE) side, and meanwhile run the non-confidential code on a JVM on the Rich Execution Environment (REE) side. The two sides interact with each other, protecting confidential computing tasks and as well keeping the Trusted Computing Base (TCB) size small.**

We implement Lejacon and evaluate it against OcclumJ (a state-of-the-art JVM-in-enclave solution) on a set of benchmarks using the BouncyCastle cryptography library. The evaluation results clearly show the strengths of Lejacon: it achieves competitive performance in running Java confidential code in enclaves; compared with OcclumJ, Lejacon achieves speedups by up to 16.2× in running confidential code and also reduces the TCB sizes by 90+% on average.

***Index Terms*—Software Guard Extensions, Separation Compilation, Native Confidential Computing Service, Runtime, Secure Closed-World**

I. Introduction

As companies are deploying their applications on clouds, security concerns usually exist in running these applications on non-confidential platforms. Confidential computing provides trusted execution environments (TEEs) in which confidential code, data, and execution are protected, meeting the security requirements for computations [1].

Software Guard Extensions (SGX) is a typical confidential computing technology developed by Intel [2]. It is the cornerstone of the confidential computing roadmap for Intel server processors (e.g., Xeon processors) [3], and attracts many attentions from cloud service providers such as Microsoft [4] and IBM [5]. More specifically, SGX provides users with *hardware enclaves*. An enclave is an execution environment with a reserved memory region, guaranteeing confidentiality and integrity of an application's code and data—in the SGX threat model, only the CPUs and the enclaves are trusted [6], [7], [8]. A software application can then be divided into confidential and non-confidential code: on the Rich Execution Environment (REE) side, the non-confidential code runs normally; on the TEE side, the confidential code runs in an enclave, isolated from the users' operating systems, the off-chip hardware systems, and the other software applications in the non-confidential world. Here we use the term "confidential code" to indicate that the execution and the data are protected, as SGX provides a form of isolation for code and data [9].

Many efforts, including library operating systems (LibOSes) [10], [11], [12], [13], [14] and partitioning frameworks [15], [16], have been spent on deploying and running C/C++ applications in enclaves. On the other hand, numerous cloud applications are written in Java, while Java is vulnerable due to its flexibility. For example, attackers can dynamically load a malicious class to read decrypted secrets in the Java heap—this is how the famous Log4j attacks do [17]. Demand for running Java applications in enclaves thus exists [18], [19], whilst it is challenging to support Java confidential computing—the traditional SGX technique is not Java-friendly, since enclaves can only be created by C/C++/native APIs and thus Java is inherently not supported in the enclave [20].

One mainstream is to run Java applications on Java virtual machines (JVMs) running in enclaves [20]. SGX-LKL-JVM [21], Civet [22], Uranus [23] and Occlum-JVM [24] belong to this mainstream. Meanwhile, these *JVM-in-enclave* solutions still suffer from serious limitations. As Fig. 1 shows, a JVM (*e.g.*, OpenJDK's HotSpot [25] or Eclipse's OpenJ9 [26]) needs to be deployed in the enclave, deepening the computation stack; all code residing in the enclave, including the confidential Java code, the JVM, and the libraries, is all taken into the Trusted Computing Base (TCB) for the application. The larger the TCB size, the larger the attack surface, the more likely the computing is vulnerable [27]. In

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00142

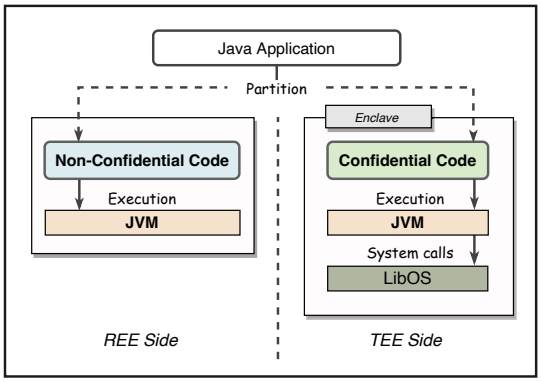


Fig. 1. A typical JVM-in-enclave solution for Java confidential computing.

this respect, the attack surface of the enclave is significantly bloated by a JVM and its libraries that usually have millions of LoC (lines of code).

An alternative is to compile the whole application into native code and run it in an enclave, which allows JVMs to be eliminated from enclaves. However, it is not feasible in practice. *First*, a great number of system and user libraries also need to be compiled and pre-deployed. These libraries reside in the enclave, incurring a large attack surface. *Second*, an enclave is only an execution environment, rather than a fully functional operating system. An application may have to run non-confidential code and perform system calls on the REE side.

Only the confidential code, rather than the whole application, needs to run in the enclave. Having realized the potentials of Java confidential computing and the limitations of existing JVM-in-enclave solutions, we propose *Lejacon*, a lightweight and efficient approach to Java confidential computing: Lejacon compiles the confidential code into a bundle of Native Confidential Computing (NCC) services; only the NCC services, along with the classes they depend on, are ultimately executed in enclaves; the JVM, the non-confidential code and its libraries, are still executed on the REE side.

This paper makes the following contributions:

- Principle. We propose and formalize a Secure Closed-World (SCW) principle. This principle is employed to direct Java confidential computing, allowing all of the reachable classes, rather than gigantic libraries, to be natively compiled. This principle is general and can also be applied, with slight modifications, to other partition frameworks for confidential computing.
- Approach. Lejacon takes a balance between Java's platform independence and the necessity of native execution in SGX enclaves: it follows a demand-driven programming paradigm, allowing programmers to annotate in a Java application the confidential computing tasks; it takes an SCW-directed separate compilation technique, which compiles the confidential code into NCC services and the non-confidential code into bytecode files; Lejacon also provides a runtime system for running Java applications and their NCC services.
- Implementation and Evaluation. We implement Lejacon as a lightweight infrastructure for Java confidential computing. We also evaluate it against OcclumJ (a state-of-the-art JVM-in-enclave solution) on a set of benchmarks using the BouncyCastle cryptography library. The evaluation results clearly show the strengths of Lejacon: it achieves competitive performance in running Java confidential code in enclaves; compared with OcclumJ, Lejacon achieves the speedups by up to 16.2× in running confidential code and also reduces the TCB sizes by 90+% on average.

To the best of our knowledge, Lejacon is the first technique that applies the SCW principle to Java confidential computing; Lejacon significantly reduces developers' costs, as Java developers may benefit from Lejacon in using pre-deployed confidential cloud services and developing their own confidential services. Lejacon is also valuable in practice: it provides the fundamental approach to the Teaclave Java TEE SDK and has been employed in practice. We envision Lejacon, along with the SDK, to provide best practices for Java confidential computing.

The remainder of this paper is organized as follows: Section II introduces the Intel's SGX technique. Section III uses an example to illustrate how Lejacon supports Java confidential computing. Section IV presents the technique details of Lejacon. Section V evaluates Lejacon. Section VI presents related work and Section VII concludes.

II. Background

SGX is an x86-64 instruction set extensions technique. Given a software application, SGX supports its confidential computing as follows.

First, the application is split into the confidential and the non-confidential parts, which will run on the TEE side and the REE side, respectively.

Second, when a confidential computing task needs to be performed, the application launches an enclave. An enclave is encrypted in a special area of the physical memory. Data in the enclave is decrypted only inside the CPU and only at the request of instructions executed from within the enclave itself.

Third, at runtime, for performing a confidential computing task, SGX constructs a memory heap and a context in the enclave [28]—the former is leveraged for in-enclave memory allocation; the latter corresponds to the control structure and the stack of each process/thread entering the enclave. In addition, only the code within the enclave can access its data; when the task completes, its data will reside in the enclave or be destroyed.

Fourth, to bridge the confidential/non-confidential parts, SGX provides two interfaces: Enclave Call (ECall) for the calls from the non-confidential world to the functions within enclaves, and Outside Call (OCall) the calls in reverse [29]. An ECall typically invokes a confidential computing task and retrieves the result; an OCall either invokes a user-defined,

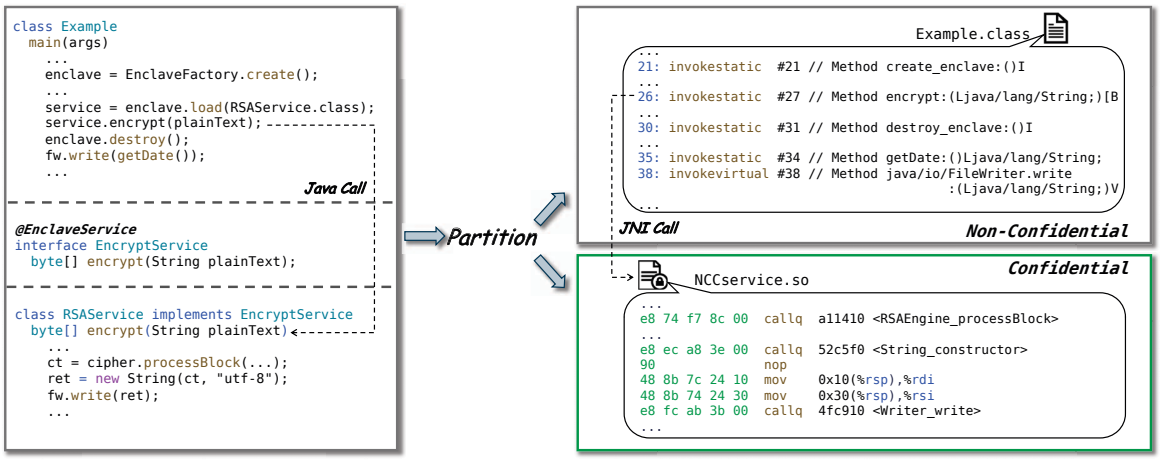


Fig. 2. An example illustrating how Lejacon partitions an application for confidential computing.

non-confidential function outside the enclave or performs a system call. Enclaves can also collaborate with each other after attestation—an enclave must prove its trustworthiness to the counterpart. For example, an enclave may establish a secure channel with other enclaves based on Diffie-Hellman Key Exchange for protecting subsequent communications [30], [31], [32].

III. An Illustrative Example

Fig. 2 shows an illustrative example of using Lejacon to support Java confidential computing. A developer annotates the interface `EncryptService` using `@EnclaveService`, specifying a confidential method `encrypt`. `RSAService` is an RSA encryption service implementing `EncryptService`. Thus its method `encrypt` needs to be confidentially executed. In addition, the developer needs to create an enclave (*i.e.*, `EnclaveFactory.create()`), load and execute the method `encrypt` (*i.e.*, `service.encrypt(...)`), and destroy the enclave after the confidential method completes (*i.e.*, `enclave.destroy()`).

Lejacon automatically divides the application into two parts. Lejacon performs reachability analysis of the confidential service `RSAService`, and compiles the class, along with the classes it depends on, into `NCCservice.so`. Here `NCCservice.so` is a shared library containing an NCC service `RSAService` and its dependent classes. The non-confidential part is compiled into bytecode that can invoke the NCC service using the Java Native Interface (JNI) technology [33].

At runtime, the application still runs on a JVM on the REE side. Once a confidential computing task `RSAService.encrypt(...)` needs to be executed, an enclave is created and initialized (*i.e.*, `create_enclave()`). Lejacon deploys `NCCservice.so` on the TEE side and executes the NCC service in the enclave. An NCC service can be invoked either by the non-confidential code or by the confidential code—Once invoked, the service will run, and may invoke any native functions in `NCCservice.so`. After the NCC service completes, Lejacon calls a native function `destroy_enclave()` to shut down the enclave and release the computing resources. Note that it is a design choice that we let developers create/destroy enclaves, since (1) the compiler is not sensitive to the availability of enclaves, and (2) developers should be aware of where their code is running.

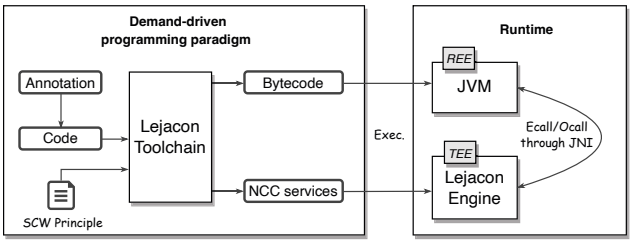


Fig. 3. An overview of Lejacon.

IV. Approach

As Fig. 3 shows, given a Java application, Lejacon supports its confidential computation by (1) a demand-driven programming paradigm and a separate compilation technique that divide the application into the confidential and the non-confidential parts (§IV-B), and (2) a runtime system that runs the confidential code on a JVM on the REE side and runs the NCC services in enclaves (§IV-C).

A. Foundation

A Java method is protected if it runs in an enclave. However, the method can still be under attack if it invokes any methods or APIs (directly or indirectly) from the non-confidential world. Therefore, Lejacon directs Java confidential computing by following a Secure Closed-World (SCW) principle.

Proposition 1 (SCW Principle)**.** A confidential service, along with all of the classes it depends on and all of the methods it

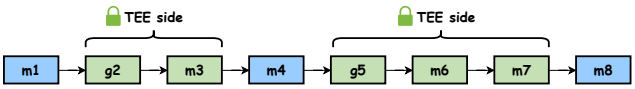


Fig. 4. An example of a call chain with two confidential call subchains. On the REE side, m_1, m_4, and m_8 run successively, while m_1 calls g_2 and m_4 calls g_5.

invokes, must be executed in the enclave(s); it does not invoke any methods from the non-confidential world.

Formally, we define confidential computing from the perspective of call chains.

Definition 1 (Call chain)**.** A call chain is defined as a sequence of method invocations: $[m_0 \rightarrow m_1 \rightarrow \cdots \rightarrow m_n]$. For representing method invocations, we use $m[\ldots g \ldots]$ to denote that m invokes a method g.

Definition 2 (Confidential call chain)**.** A confidential call chain is a chain $s : cm[\ldots]$, where cm is a *starting confidential method*. Any method in $[\ldots]$ also needs to be confidentially computed.

As Fig. 4 shows, $s_1 : g_2[m_3]$ and $s_2 : g_5[m_6[m_7]]$ are two confidential call subchains, respectively. The two subchains need to run in enclaves.

Given any confidential method cm, we then use the SCW principle to construct a set (say SC) that contains all reachable classes (say *confidential classes*) for running cm's confidential call subchain. Let CM be the set of confidential methods, C the set of classes, and M the set of class methods. The SCW principle can be formalized into a set of rules:

$$\forall cm \in CM, m \in M, reach(cm, m) \Rightarrow m \in CM; \quad \text{(RULE 1)}$$

$$\forall sc \in SC, c \in C, reach(sc, c) \Rightarrow c \in SC; \quad \text{(RULE 2)}$$

$$\forall cm \in CM, c \in C, reach(cm, c) \Rightarrow c \in SC; \quad \text{(RULE 3)}$$

$$\forall sc \in SC, m \in M, reach(sc, m) \Rightarrow m \in CM, \quad \text{(RULE 4)}$$

where

$$reach(A, B) = \begin{cases} true & \text{if a class/method } B \text{ is reachable} \\ & \text{from another class/method } A; \\ false & \text{otherwise.} \end{cases}$$

That is, any method invoked in a confidential method or any class reachable from a confidential method/class is also confidential. A detailed explanation of the reachability will be presented in §IV-B2.

B. Programming Paradigm

1) Annotation: Lejacon follows a demand-driven programming paradigm that developers explicitly annotate confidential computing tasks and then Lejacon separately compiles these tasks into NCC services.

In order to do this, a developer creates a service interface with an annotation `@EnclaveService` and then implements the service's methods in each implementation class. As the following code shows, an `@EnclaveService` interface (*e.g.*, `Service`) is taken as an interface for enclaves; the implementation class `ServiceImpl` and its two methods (*i.e.*, `foo`, `bar`) are confidential.

```
1 @EnclaveService
2 interface Service{
3   void foo();
4   byte[] bar(String p)
        ;
5 }
```

```
1 class ServiceImpl
2   implements Service{
3     void foo(){...}
4     byte[] bar(String p)
        {...}
5   }
```

Each class implementing an `@EnclaveService` interface is taken as the starting point of the reachability analysis (§IV-B2). The annotation thus directs the generation of a closed-world for Java confidential computing.

Note that to better understand application-specific demand, we require the user to annotate which service(s) should be confidential. It is a lightweight task. There remains a promising direction regarding whether confidential code can be automatically annotated.

2) Separate Compilation: The annotated interface declares confidential methods. Lejacon builds a compiler atop GraalVM [34]. GraalVM allows a Java application to be ahead-of-time (AOT) compiled into a native executable. On the basis of the GraalVM's capability of AOT compilation, Lejacon compiles all of the confidential classes and methods into a shared library. The library, which contains the compiled code of the reachable Java bytecode, is deployed on the TEE side. This separate compilation greatly reduces the TCB size.

First, Lejacon detects all of the starting confidential methods followed by performing reachability analysis. Any method declared in an `@EnclaveService` interface, if loaded by an enclave, is compiled into an NCC service. Lejacon compiles all of the reachable classes and methods into a shared library.

Algorithm 1 describes the process of reachability analysis and library generation. This algorithm takes a set of starting confidential methods as input and generates a shared library lib as output. These confidential methods are taken as the starting points for reachability analysis.

Let SC and CM be two sets storing confidential classes and methods, respectively. Lejacon leverages the static analyzer of GraalVM [35], which iteratively picks up elements from SC and CM and then resolves reachable methods and classes (lines 4∼10):

- If a class is resolved, Lejacon also resolves its parent class, its fields' types, its class initialization method ($\langle clinit \rangle$), and its object initialization methods ($c.init_0, c.init_1$, etc.) (lines 12∼18);
- If a static method is resolved, Lejacon also resolves its local variables' types and the methods invoked in this method (lines 24∼33);
- If a non-static method is resolved, Lejacon also resolves its class, its local variables' types, and the methods invoked in this method (lines 22∼33).

Algorithm 1: An algorithm for native compilation

Input : Source code of a Java application;
A set of starting confidential methods CM
Output : A native library lib

$SC_1 \leftarrow$ a set of interfaces declaring the methods in CM
$SC_2 \leftarrow$ a set of classes implementing the methods in CM
$SC \leftarrow SC_1 \cup SC_2$
repeat
  pick up e in $CM \cup SC$
  if $e \in SC$ **then** `ResolveClass(`e`)`
  else `ResolveMethod(`e`)`
until *all classes in* SC *and all methods in* CM *are resolved*
compile classes in SC and methods in CM into lib
return lib

Function `ResolveClass(`c`)`**:**
  if $\neg c.resolved$ **then**
    $c.resolved \leftarrow true$
    $SC \leftarrow SC \cup \{c.parent\}$
    $CM \leftarrow CM \cup \{c.\langle clinit \rangle\} \cup \{c.init_0, c.init_1, \dots\}$
    foreach *field* f *of* c **do**
      $C' \leftarrow Points\text{-}to(f)$
      $SC \leftarrow SC \cup C'$

Function `ResolveMethod(`$c.m$`)`**:**
  if $c.m$ *is safe and* $\neg c.m.resolved$ **then**
    $c.m.resolved \leftarrow true$
    if m *is not static* **then**
      $SC \leftarrow SC \cup \{c\}$
    foreach *variable* v *used in* $c.m$ **do**
      $C' \leftarrow Points\text{-}to(v)$
      $SC \leftarrow SC \cup C'$
    foreach *method invocation in* $c.m$ **do**
      if *a static method* $c'.m'$ *is invoked* **then**
        $CM \leftarrow CM \cup \{c'.m'\}$
      if *a non-static method* $o.m'$ *is invoked and* o *is the receiver object* **then**
        $C' \leftarrow Points\text{-}to(o)$
        foreach c' *in* C' **do**
          $CM \leftarrow CM \cup \{c'.m'\}$

Lejacon needs to perform points-to analysis (*i.e.*, $Points\text{-}to$ in Algorithm 1) [36], [37], [38] for analyzing an object's class(es) or the method(s) invoked when dynamic binding occurs.

Lejacon uses GraalVM Native Image tool [39], a native compiler for Java program, to compile all of the classes in SC and create a native library `lib`. For reducing the TCB size, Lejacon partially compiles each reachable class—any method of a confidential class, if unreachable from the starting confidential methods, can be excluded from the compilation.

Second, Lejacon uses the traditional Java programming language compiler to compile the application's non-confidential code into bytecode files (`*.class`).

Note that Lejacon does provide a set of APIs for enclaves' lifecycle management: before invoking an NCC service, it creates an enclave isolation; confidential services can then be loaded and executed inside the enclave; after completing all the confidential computations, it can destroy the enclave to recycle the memory resource. Correspondingly, a bundle of native interfaces, as the following shows, are compiled into bytecode for creating/destroying an enclave, loading an NCC service, etc.

```
static native int create_enclave();
/* an NCC function */
static native int confidential_foo(int m);
static native int destroy_enclave();
/* loads the library */
System.load("NCCservice.so");
create_enclave();
destroy_enclave();
```

3) Optimization: An NCC service is a round-trip service, indicating the time spent on confidential computing includes not only the time on in-enclave computation but also that on creating/destroying enclaves, synchronizing execution contexts, etc. (see §IV-C).

However, an enclave may be frequently entered and exited because an NCC service needs to be repeatedly invoked. It is likely to lead to heavy workloads in synchronizing execution contexts on the both sides. In case that an NCC service s needs to be invoked for n times, we have

$$ET = ET_0 + ET_1 + ET_2$$
$$ET_0 = n \times et_0$$
$$ET_1 = n \times et_1$$

where ET is the total time spent on confidential computing; ET_0, ET_1, and ET_2 are the time on running confidential code, preparing/synchronizing execution contexts, enclave management (*e.g.*, creating an enclave and loading libraries), respectively; et_0 is the time on executing s, and et_1 that on preparing the context of s.

Optimization can be conducted to reduce the time for context synchronizations. As Listing 1 and Listing 2 show, by inlining a loop containing confidential methods, Lejacon can automatically transform the loop into an NCC service. The total execution time is reduced to

$$ET = n \times et_0 + et_1 + ET_2$$

This optimization strategy saves the costs for context synchronizations, and thus reduces the total execution time. A further evaluation will be given in §V-C.

C. Runtime Support

Lejacon provides a runtime system for confidential computing. It allows for multiple enclaves, supporting much more flexible confidential computing — ideally, one enclave corresponds to one confidential call chain. As Figure 5(a) shows, the runtime is composed of:

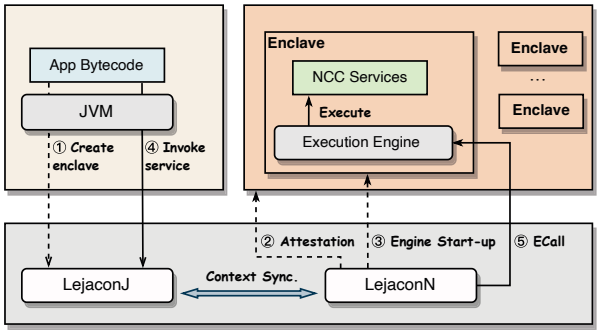
- A JVM for executing non-confidential code on the REE side. The JVM also allows the application to invoke NCC services via JNI;
- A set of execution engines, each of which is bounded to an enclave and responsible for executing NCC services in the enclave. The engine is built atop Substrate VM runtime, a popular runtime for native Java executables [40];

Listing 1. Before optimization: a service encode is repeatedly invoked.

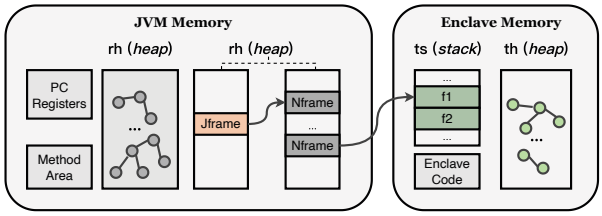
```
// main()
 ...
 String s = ...;
 for(int i=0;i<n;i++)
  result=service.
       encode(s);
 ...

@EnclaveService
interface MyService{
 String encode(String
       str);
}

class ServiceImpl1
  implements MyService{
 String encode(String
       str){...}
}
```

Listing 2. After optimization: a wrapped service wencode is invoked.

```
//main()
 ...
 String s = ...;
 result=service.
       wencode(n,s);
 ...

@EnclaveService
interface MyService{
 String encode(String
       str);
 String wencode(int n,
       String s);
}
class ServiceImpl1
  implements MyService{
 String encode(String
       str){...}
 String wencode(int n,
       String s){
  ...
  for(int i=0;i<n;i++)
   result=encode(s);
  return result;
}}
```



(a) Architecture.



(b) Memory layout.

Fig. 5. An overview of Lejacon's runtime system.

- A communication component for conducting attestation of the TEEs, managing enclaves, and creating/synchronizing execution contexts between the two sides. Specially, Lejacon uses Java dynamic proxy to delegate users' enclave service invocations to actual TEE invocations.

1) Memory Management: Memory allocation can occur on either the TEE side or the REE side. Fig. 5(b) shows the memory layout for running a Java application. The memory space is mainly managed by the JVM. However, when an NCC service needs to run, an enclave space, which is taken as a virtual space of the JVM's memory space, is set up.

For facilitating our discussions, we use rh and rs to denote the heap and the stack on the REE side, respectively. We also use th and ts to denote the heap and the stack on the TEE side, respectively.

Memory allocation can be performed during executions. Objects created by each NCC service are allocated in th. The heap is managed by a garbage collector (GC) provided by the Substrate VM runtime—the GC allocates memories in th, and collects garbage when th is full.

Similar to JVM, an enclave also has a stack space for running NCC services [28]. Stack frames of confidential calls are stored in the protected stack ts. When an ECall of an NCC service $cm[\dots]$ is made, a stack frame for cm, ϕ_{cm}, is constructed and pushed into ts; new stack frames are further pushed into ts when native functions are called.

2) Execution Context: We let an execution context be abstracted for an NCC service s.

Definition 3 (Execution context)**.** Let γ be a specific execution environment. The execution context of a method m on γ, say $\xi(m,\gamma)$, is a pair

$$\xi(m,\gamma) = \langle \gamma(\tau_m), \gamma(\phi_m) \rangle,$$

where $\gamma(\tau_m)$ is a set of heap objects on which m depends and $\gamma(\phi_m)$ is the stack frame for running m on γ.

Similarly, the execution context of an NCC service $s : cm[\dots]$ on γ, say $\xi(s,\gamma)$, is a pair

$$\xi(s,\gamma) = \langle \gamma(\tau_s), \gamma(\phi_{cm}) \rangle,$$

where $\gamma(\tau_s)$ is a set of heap objects on which the methods in s depend and $\gamma(\phi_{cm})$ is the stack frame for running cm on γ.

Here $\gamma(\tau_m)$ is computed by analyzing the objects that may be accessed by m. Similarly, $\gamma(\tau_s)$ is computed by analyzing the objects that may be accessed either by cm or by any method in $[\dots]$. The algorithm is omitted because it is similar to the serialize algorithm presented in Algorithm 2.

3) Context Synchronization: Lejacon's runtime is correct if given the same execution context, the NCC service $s : cm[\dots]$ runs equivalently in an enclave and on a JVM. Ideally, an execution context equivalent to that on the JVM should be prepared for executing s in an enclave, *i.e.,*

$$\xi(s,tee) = \xi(s,ree)$$

For guaranteeing the equivalence, the communication component is responsible for synchronizing contexts. TABLE I lists the APIs for managing execution contexts. These APIs allow NCC services to run on TEEs while keeping the contexts on both sides synchronized. For example, for executing methods in an NCC service s, Lejacon creates the execution context using `createTEEContext`: $\xi(s,ree)$ is collected and serialized, and then transmitted to the TEE

TABLE I
APIs FOR CONTEXT SYNCHRONIZATION.

API	Description
`Context createTEEContext()`	Create $\xi(m, tee)$ using $\xi(m, ree)$.
`void updateTEEContext()`	Update $\xi(m, tee)$ using $\xi(m, ree)$.
`void updateREEContext()`	Update $\xi(m, ree)$ using serializable objects exposed by m.
`Context getTEEContext()`	Retrieve serializable objects exposed by m.
`Context getREEContext()`	Retrieve $\xi(m, ree)$.

Algorithm 2: An algorithm for context synchronization

```
   Input  : An invoked confidential method m
   Output : A set of objects in synchronization Context
1  unscanned ← a set of objects referenced by static class variables
     and variables in the stackframe of m
2  scanned ← updated ← ∅
3  repeat
4      o ← pick(unscanned)
5      scanned ← scanned ∪ {o}
6      if o.serializable then
7          foreach referenced object o' of o do
8              if o' ∉ (scanned ∪ unscanned) then
9                  unscanned ← unscanned ∪ {o'}
10 until unscanned = ∅
11 foreach o ∈ scanned do
12     so ← serialize(o)
13     Context ← Context ∪ {so}
```

side; the TEE side receives and deserializes $\xi(s, ree)$ followed by setting up $\xi(s, tee)$, the execution context for s on the TEE side. Lejacon can also update execution contexts using `updateTEEContext` or `updateREEContext`, letting an execution context be transmitted for updating the execution context on the other side.

As Algorithm 2 shows, given an NCC service, Lejacon detects which objects in rh it relies on. It is much like the *mark-and-sweep* algorithm used for garbage collection [41], [42], [43]: all of the objects that are reachable from the roots (*i.e.*, the variables in $\gamma(\phi_m)$ and the static class variables) are collected. Since these objects need to be transmitted between the JVM and the enclave, unserializable objects are excluded from object collection (lines 6∼9).

Lejacon takes a *passing-by-value* strategy to synchronize contexts. Thus it needs to serialize and transmit the heap objects between the REE and the TEE side when necessary (lines 11∼13). After deserialization, the context on one side is used for updating the context on the other side. Objects can be missing on one side if they are non-serializable; Lejacon fails to run a method (or an NCC service) if the method (or the service) requires to access any missing object(s).

4) Enclave/Outside Calls: A Java application can invoke a native function inside the enclave through JNI. Lejacon sets up two agents: LejaconJ and LejaconN. LejaconJ runs on the JVM. Every time the application needs to perform confidential computing, LejaconJ sends a request to LejaconN. LejaconN is an agent of the NCC services and can further invoke the services in the enclave through ECalls.

Lejacon does not support application-level OCalls, since the SCW principle requires that all methods reachable from a confidential method must be executed in the enclave(s). Meanwhile, an enclave is in the user mode. Thus the system services (*e.g.*, process creation and management, device handling) can only be requested via explicit OCalls (system calls),[1] even though it is often not necessary to synchronize contexts for these OCalls.

5) Attestation: Lejacon performs a standard remote attestation when using an enclave [32]. The remote attestation guarantees the trustworthiness of the enclaves, and as well the confidentiality and integrity of the confidential tasks running in them.

6) Put It Together: Fig. 5(a) shows the process of running a confidential application on Lejacon. When the application needs to create an enclave or call a confidential service, it sends a confidential request to LejaconJ (① or ④). LejaconJ parses the request, serializes the execution context, and transmits the context to LejaconN. In order to use an enclave, LejaconN makes an ECall to build an enclave (⑤) and performs a remote attestation on the enclave (②). In order to invoke an NCC service, LejaconN sets up the execution context on the TEE side. The execution engine in the enclave then starts, performing confidential computing (③ and ⑤). After that, the output and the updated execution context on the TEE side are serialized and transmitted to LejaconJ. LejaconJ retrieves the results and updates the contexts on the REE side.

D. Infrastructure

We have implemented Lejacon as a lightweight infrastructure for Java confidential computing. Lejacon supports the development of confidential Java applications on the basis of Intel SGX SDK [44]. It provides an annotation system for supporting the demand-driven programming paradigm, a toolchain built atop Native Image tool [39] for Java native compilation, and a runtime (and supporting libraries) built atop Substrate VM runtime [40] and OpenJDK [45]. Lejacon uses Intel's Data Center Attestation Primitives (DCAP) [46] to support remote attestations when creating enclaves.

V. EVALUATION

We have evaluated Lejacon on a set of benchmarks. Our evaluation is designed to answer three research questions:

- **RQ1** Can Lejacon support Java confidential computing effectively?
- **RQ2** Can Java applications achieve competitive performance on Lejacon, compared with some state-of-the-art runtime system(s)?
- **RQ3** Can Lejacon be practically used in running Java confidential computing tasks?

[1] In practice, system calls are not in the closed-world. However, Lejacon allows for explicit OCalls in a confidential call chain that otherwise its functionalities are constrained. Implicit OCalls, i.e., application-level method invocations from TEE to REE, are forbidden.

TABLE II
TECHNIQUES UNDER COMPARISONS.

Technique	Is the app partitioned?	REE Side		TEE Side	
		Use JVM	Code Rep.	Use JVM	Code Rep.
BaseJ	×	✓	bytecode	\	\
OcclumJ	✓	✓	bytecode	✓	bytecode
Lejacon	✓	✓	bytecode	×	native code

A. Setup

Technique. We compare Lejacon against two techniques: BaseJ and OcclumJ.

1) BaseJ is the baseline. It corresponds to a general scenario, in which each application runs only on a JVM, without performing any confidential computing.
2) OcclumJ is a JVM-in-enclave, LibOS-based solution. It uses Occlum (a library OS [14]) as the underlying system for running confidential code on a feature-complete JVM in the enclave; the non-confidential code communicates with the confidential code through remote method invocations (RMIs).

TABLE II summarizes the characteristics of each technique, where the code can be either in bytecode or in native.

To the best of our knowledge, OcclumJ is the most appropriate technique that is comparable with Lejacon. Some other techniques, such as Civet [22], Uranus [23] and Montsalvat [47], are not compared in our study, since their tools are not publicly available or they perform platform-dependent compilations of Java confidential/non-confidential code.

Dataset. We prepare a dataset for our evaluations. TABLE III shows an overview of the dataset. It contains four mini applications (`app-print`, `app-digest`, `app-rsa`, and `app-sqlparser`), each containing 1∼8 starting confidential methods. It also contains six CoreTest (ct) benchmarks (`ct-asn1`, `ct-i18n`, `ct-util`, `ct-math`, `ct-pqc` and `ct-crypto`) that are synthesized from 254 tests for the core module of the BouncyCastle, a widely-used open source cryptography library [48]; each benchmark invokes 2∼168 starting confidential methods (*i.e.*, BouncyCastle APIs).

Metric. We use three metrics in the evaluation:

(1) **TCB size**. We measure the TCB size of each benchmark by ① *TCB-IC* (Number of in-enclave Java classes), ② *TCB-IM* (Number of in-enclave methods), and ③ *TCB-LS* (Size of the built library). The larger the TCB size, the larger the attack surface, and the more vulnerable an application's confidential computing [27].

(2) **IEMF** (In-Enclave Memory Footprint). It measures at runtime the in-enclave memory footprint of each benchmark. The larger the memory footprint, the more likely the confidential computing is vulnerable and the more computation resources are consumed for its confidential computing.

(3) **ET** (Total Execution Time). It measures the performance using the execution time of the benchmarks. As §IV-B3 explains, the execution time is further divided into ① ET_0 (Time on running confidential code), ② ET_1 (Time on context synchronization), ③ ET_2 (Time on environment management).

As for BaseJ, ET of each benchmark only contains the time spent on running confidential code on a non-confidential JVM. As for OcclumJ, ET_2 also contains the time spent on JVM management (*i.e.*, starting a JVM in an enclave).

Configuration. Our evaluation is conducted on a server equipped with Intel Xeon Platinum 8369B CPU (2.70GHz), 30GB DRAM, and Ubuntu 18.04.5 LTS. The SDK for the Intel SGX platform [44] is of version 2.15. OpenJDK v11 [45] is used on the REE side. In addition, Lejacon employs the Native Image tool of GraalVM CE v21.3.0 [39] to generate native libraries; the OcclumJ executables (jar files) are created for Occlum v0.24.0.

B. Confidentiality

TCB size. TABLE IV summarizes the TCB sizes of benchmarks. It can be observed that the applications on Lejacon are of much smaller TCB sizes than those on OcclumJ. Compared with OcclumJ, Lejacon reduces the TCB sizes by 90+%, with fewer in-enclave classes and methods.

One main reason is that OcclumJ creates, for each benchmark, a fat jar file containing third-party libraries. When running on OcclumJ, each benchmark (even the simplest application `app-print`) needs to be zipped into a jar file of hundreds or thousands of classes. For example, `app-sqlparser` uses a library Druid [49], and its jar file contains about 2.1K Java classes and 32K methods; `app-digest` and `app-rsa` depend on the whole BouncyCastle library, and each of their jar files is of 4.7K classes and 30K+ methods. The ct benchmarks also rely on BouncyCastle, and each jar file is of 3.1K classes and 23K methods.

Comparatively, after reachability analysis, Lejacon excludes a large number of unreachable classes and methods, reducing the numbers of in-enclave classes and methods by 42∼93% and 61∼99%, respectively. The reachability analysis significantly reduces the numbers of in-enclave classes and methods.

The difference in the TCB size also arises due to the runtime systems (OcclumJ's Java runtime and Lejacon's runtime) and libraries introduced into enclaves. For OcclumJ, *TCB-LS* for the simple `app-print` is 382MB, about 23.9× of that in Lejacon. A small-sized system library (<10MB), rather than the rich libraries for the whole Java runtime, is compiled into each Lejacon's executable; it significantly shrinks the attack surface.

Memory footprint. TABLE IV shows the in-enclave memory footprint of each benchmark. Obviously, Lejacon also keeps much smaller memory footprints than OcclumJ. The memory footprints *w.r.t.* the four mini applications and most of the ct benchmarks can be reduced by 90+%. The memory footprint *w.r.t.* `ct-crypto` on Lejacon is reduced by 72.5%, compared with that on OcclumJ.

The main reason is that Lejacon does not take the JVM-in-enclave strategy. Thus the memory for the Java runtime (300+MB per benchmark) is waived. The reachability analysis

TABLE III
BENCHMARKS USED IN THE EVALUATION.

Benchmark	Main Third Party Library	#Starting Confidential Methods	Description
app-print	\	1	A simple program that prints strings on the terminal.
app-digest	BouncyCastle-full	8	An application that supports hash algorithms.
app-rsa	BouncyCastle-full	5	An application that supports RSA, an asymmetric encryption algorithm.
app-sqlparser	Druid	3	A SQL parser using Druid [49] (a Java library for database connection pools).
ct-asn1	BouncyCastle-core	48	Tests for parsing and writing asn.1 objects.
ct-i18n	BouncyCastle-core	2	Tests for the internationalization APIs.
ct-util	BouncyCastle-core	5	Tests for the utility classes.
ct-math	BouncyCastle-core	19	Tests for the APIs in the math package.
ct-pqc	BouncyCastle-core	98	Tests for the APIs in the post-quantum lightweight crypto packages.
ct-crypto	BouncyCastle-core	168	Tests for the base classes from the crypto APIs.

TABLE IV
COMPARISON OF THE TCB SIZES AND MEMORY FOOTPRINTS BETWEEN OCCLUMJ AND LEJACON. HERE ∇ IS USED TO PRESENT THE PERCENTAGES OF DECREASES OF LEJACON, COMPARED WITH OCCLUMJ, IN ***TCB-LS*** AND **IEMF**; app-digest RUNS SEVERAL HASH ALGORITHMS (*e.g.*, MD5) USING A TOTAL OF 256KB MESSAGES.

Benchmark	**OcclumJ** (w/o reachability analysis)				**Lejacon** (w/ reachability analysis)			
	TCB-IC	***TCB-IM***	***TCB-LS* (MB)**	**IEMF (MB)**	***TCB-IC***	***TCB-IM***	***TCB-LS* (MB) (∇)**	**IEMF (MB) (∇)**
app-print	436	3285	382	351.4	212	7	16 (95.8%)	10.5 (97.0%)
app-digest *(256KB)*	4757	32450	388	407.1	367	608	17 (95.6%)	25.1 (93.8%)
app-rsa	4757	32456	388	404.0	614	3209	21 (94.6%)	25.4 (93.7%)
app-sqlparser	2112	31725	386	355.4	1221	12234	29 (92.5%)	15.7 (95.6%)
ct-asn1	3128	23279	429	360.6	578	2382	21 (95.1%)	24.6 (93.2%)
ct-i18n	3128	23279	429	351.4	218	30	16 (96.3%)	10.5 (97.0%)
ct-util	3128	23279	429	351.4	218	47	16 (96.3%)	11.5 (96.7%)
ct-math	3128	23279	429	539.4	534	3015	21 (95.1%)	75.3 (86.0%)
ct-pqc	3128	23279	429	545.1	409	1406	22 (94.9%)	97.3 (82.2%)
ct-crypto	3128	23279	429	547.9	1346	8644	33 (92.3%)	150.4 (72.5%)

taken by Lejacon also helps exclude many unnecessary classes, reducing the code residing in the enclave memory.

Answer to RQ1: Lejacon performs reachability analysis and leverages a lightweight runtime system, reducing the TCB sizes by more than 90% and keeping smaller memory footprints than OcclumJ at runtime.

C. Performance

TABLE V and Fig. 6 compare the total execution time of the NCC services of the benchmarks. Benchmarks running on OcclumJ and Lejacon are in general slower than those on BaseJ. For example, BaseJ outperforms Lejacon and OcclumJ by 2× and 28×, respectively, in its execution time when running app-sqlparser. Each benchmark, when running on SGX, needs an enclave. The overhead is significant (26.6∼99.8% of the total execution time on OcclumJ and 6.6∼99.9% of that on Lejacon); serialization, communication, memory encryption and decryption also introduce extra runtime overhead [50], [51]. Comparatively, BaseJ does not run confidential code in enclaves and incurs no extra cost of using enclave(s).

A further investigation helps draw out four observations. *First*, confidential code is in general fast when running on Lejacon. Compared with OcclumJ, Lejacon achieves the speedups by up to 16.2× (app-print) in running confidential code. Even though running ct-asn1 on Lejacon incurs the cost of 499.7ms in environment management, it is about 18.6% faster than that on BaseJ in its total execution time. Indeed, confidential code is AOT compiled into native code, and thus Lejacon is faster in running confidential code of most benchmarks than BaseJ and OcclumJ which interpretively execute bytecode on JVMs. Note that JVM can perform just-in-time (JIT) compilation and optimization when running computation-intensive applications/services. Thus the confidential code of ct-pqc and ct-crypto runs faster on BaseJ and OcclumJ than on Lejacon.

Second, time needs to be spent on context synchronizations and serialization. As Fig. 7 shows, ET_1 of running app-digest on Lejacon (or OcclumJ) increases along with the increase of the input's size. The main reason is that both Lejacon and OcclumJ do partition applications, incurring overheads for transmitting data across the enclaves. It imposes the necessity of reducing data transmissions and extra costs during confidential computing. Furthermore, Lejacon is faster than OcclumJ in preparing/synchronizing execution contexts — when the input size is less than 1MB, ET_1 on Lejacon is about 50% of that on OcclumJ; as the input size increases, ET_1 on Lejacon grows, while it is always less than that on OcclumJ.

Third, the optimization strategy taken by Lejacon improves its performance. As we have explained in §IV-B3, a loop of running an NCC service can be optimized into one wrapped service such that redundant context synchronizations are waived. We run app-digest with different hash algorithms for 1K times. As Fig. 8 shows, the optimization strategy helps speed up the services by 13.2∼71.9% when the NCC service is repeatedly executed.

Fourth, costs for environment management are significant. Both Lejacon and OcclumJ need to create enclaves. As TABLE V shows, for each benchmark, Lejacon needs to spend

TABLE V

EXECUTION TIME OF THE BENCHMARKS ON DIFFERENT RUNTIME SYSTEMS. Here (1) $\triangle$ measures the increase of the execution time on Lejacon or OcclumJ, compared with that on BaseJ; (2) for each benchmark, p measures how many percentage of the total execution time is spent on an activity.

Benchmark	BaseJ	OcclumJ				Lejacon			
	ET	ET($\triangle$)	ET_0 (p(%))	ET_1 (p(%))	ET_2 (p(%))	ET($\triangle$)	ET_0 (p(%))	ET_1 (p(%))	ET_2 (p(%))
`app-print`	<0.05ms	6.2s (>$10^3\times$)	<0.05ms (0.0)	14.2ms (0.2)	6.1s (99.8)	383.4ms (>$10^3\times$)	<0.05ms (0.0)	<0.05ms (0.0)	383.4ms (99.9)
`app-digest`	171.1ms	8.9s (52.0×)	382.2ms (4.3)	160.8ms (1.8)	8.4s (93.9)	964.1ms (5.6×)	105.7ms (11.0)	80.5ms (8.3)	777.9ms (80.7)
`app-rsa`	345.4ms	6.8s (19.6×)	629.2ms (9.3)	30.0ms (0.4)	6.1s (90.3)	1.1s (3.3×)	574.4ms (50.1)	2.2ms (0.2)	568.9ms (49.7)
`app-sqlparser`	230.6ms	6.5s (28.0×)	327.4ms (5.1)	28.1ms (0.4)	6.1s (94.5)	451.9ms (2.0×)	3.3ms (0.7)	0.7ms (0.2)	447.9ms (99.1)
`ct-asn1`	749.5ms	7.7s (10.3×)	1.5s (19.3)	65.7ms (0.8)	6.2s (79.8)	610.4ms (0.8×)	99.0ms (16.2)	11.7ms (1.9)	499.7ms (81.9)
`ct-i18n`	<0.05ms	6.2s (>$10^3\times$)	1.1ms (0.0)	14.0ms (0.2)	6.2s (99.8)	497.5ms (>$10^3\times$)	<0.05ms (0.0)	0.2ms (0.0)	497.3ms (99.9)
`ct-util`	2.4ms	6.1s (>$10^3\times$)	6.1ms (0.1)	17.1ms (0.3)	6.1s (99.6)	489.7ms (204.0×)	0.3ms (0.1)	1.7ms (0.3)	487.7ms (99.6)
`ct-math`	4.3s	14.5s (3.4×)	7.4s (50.7)	52.7ms (0.4)	7.1s (48.9)	8.7s (2.0×)	7.0s (80.8)	9.2ms (0.1)	1.7s (19.1)
`ct-pqc`	10.5s	20.8s (2.0×)	13.6s (65.1)	128.4ms (0.6)	7.1s (34.3)	15.6s (1.5×)	13.9s (89.2)	32.6ms (0.2)	1.7s (10.6)
`ct-crypto`	12.5s	27.5s (2.2×)	20.0s (72.8)	170ms (0.6)	7.3s (26.6)	25.8s (2.1×)	24.0s (93.2)	45ms (0.2)	1.7s (6.6)

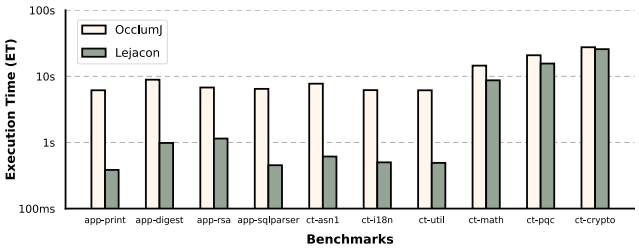


Fig. 6. Comparison of the execution time between OcclumJ and Lejacon on the benchmarks.

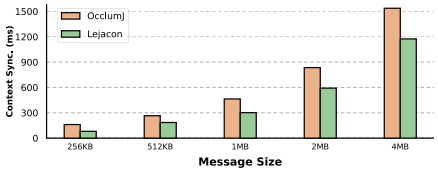


Fig. 7. ET_1 of `app-digest` with different sizes of inputs.

0.4∼1.7s on environment management and OcclumJ 6.1∼8.4s. OcclumJ spends 5.4∼7.6 more seconds on environment management than Lejacon—the time is mainly spent on initializing an enclave for running the library OS and starting a JVM for running confidential code.

Answer to RQ2: Lejacon achieves competitive performance in running Java confidential code. Lejacon speedups enclave initialization, and also takes an optimization strategy which reduces the cost of synchronizing contexts.

D. Correctness and Practical Use

We further compare the test results on different runtime systems. We run the BouncyCastle's unit tests on BaseJ and on Lejacon. A comparison of the results reveals that Lejacon is consistent with BaseJ in running these tests. Thus Lejacon is trustworthy in executing the BouncyCastle tests containing confidential computing tasks.

However, it is notable that execution contexts need to be prepared for running confidential code on Lejacon; an NCC service may fail to run if it relies on any unserializable objects on the REE side. Furthermore, SGX does not allow for parallel computing on both sides. It remains one of the

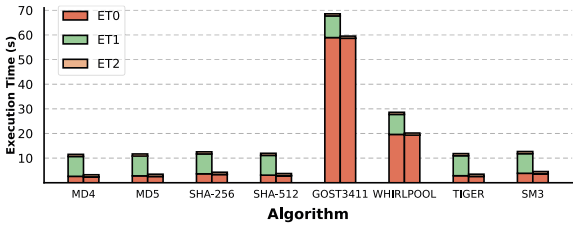


Fig. 8. Comparison of the execution time before and after optimization. Here `app-digest` with eight hash algorithms is chosen and each algorithm runs for 1K times.

TABLE VI

TYPICAL APPLICATIONS RUNNING ON LEJACON.

Application	LoC	*TCB-LS* (MB)	Description
SQLAudition	13,556	30.54	Audit the security of SQL statements to prevent malicious SQL from actual executions. We let the audit service run on the TEE side, expecting the process to be blind to the attackers and keeping the integrity of the results.
ReEncryption	32,009	19.26	Decrypt the cipher-text and encrypt it again with another key such that the cracking of the original key will not compromise the objective applications.

future directions to extend Lejacon such that it allows contexts containing unserializable objects to be synchronized and as well parallel computing to be performed on both sides.

In addition, the SCW principle might lead to false positives since Lejacon over-approximately collects reachable classes/methods. Java reflections inevitably have false negatives, as it is impossible to collect reflected classes/methods through traditional static program analysis. Correspondingly, Lejacon leverages a dynamic analysis technique, which collects reflection calls at runtime and further compile the reflected classes/methods if they are reachable from the confidential services. It remains a future direction in statically analyzing Java reflections and natively compiling confidential services of reflected classes/methods.

Lejacon is valuable in practice. It has been incorporated into the Teaclave Java TEE SDK (an open-sourced industry product) and served confidential computing in production en-

vironments. TABLE VI shows two real applications benefiting from Lejacon and the SDK.

Answer to RQ3: Lejacon is correct in running the benchmarks. However, an NCC service may fail to run on Lejacon if the execution context is not well prepared. Lejacon is also used in real production environment.

VI. Related Work

We discuss two strands of related work: (1) partition frameworks for confidential computing; and (2) runtime supports for confidential computing.

A. Partition Framework for Confidential Computing

An application is usually partitioned for confidential computing. Glamdring is a source-level partitioning framework of C applications for Intel SGX, using static program slicing [15]. Rubinov *et al.* propose an automated partitioning framework that uses taint analysis for partitioning and running Android confidential computing on ARM's TrustZone [52]. CFHider provides control flow confidential protection by moving branch statement conditions to SGX enclaves, which offers high confidentiality while maintaining low overhead [53]. Similar to Lejacon, the Montsalvat technique also leverages GraalVM Native Image tool [39] to build Java confidential applications [47]. It compiles both non-confidential and confidential code into native code and introduces proxies for objects appearing in different environments simultaneously.

Similarly, Lejacon partitions the application into the confidential and non-confidential parts. However, Lejacon keeps the non-confidential code in form of bytecode and compiles the confidential code into native services. Thus Lejacon is much more suitable for confidential serverless computing where services are short-term and sensitive to startup latency.

Lejacon employs separate compilation to partition a Java application. Separate compilation is widely used in practice. It refers to the ability that each program module can be compiled separately but linked together to produce the final executable. Many separate compilation techniques do exist for speeding up compilation [54], [55] and enabling cross-language compilation [56], [57], [58]. Some other techniques, such as [59], [60], are also available for studying how the separate compilation can be precisely conducted.

To the best of our knowledge, Lejacon is the first technique that applies the SCW principle to Java confidential computing: it performs separate compilation, in which the Java source code is separately compiled into a Java application along with a set of native services; it also performs reachability analysis during compilation such that any NCC service can run in an enclave closurely.

B. Runtime Support for Confidential Computing

Several TEE solutions do exist in recent years, supporting trustworthy/confidential computing. ARM's TrustZone is a confidential computing technique that is mainly used by IoT devices [61], [62]. It divides the CPU mode into the secure/normal modes and splits computer resources between the two modes. AMD SEV extends the AMD-V architecture in running secure virtual machines (VMs) by encrypting the VM's memory image [63]. Sanctum offers provable isolation of software modules sharing resources and provides a scheme to defend against software side-channel attacks [64]. Penglai provides a software-hardware co-design to support fine-grained, large-scale secure memory with fast-initialization [65].

One mainstream is to run confidential code on library OSes deployed on the above TEE solutions [13], [66]. SCONE provides a modified C standard library inside enclaves [10]. It forwards libc calls and system calls to the host. Graphene-SGX implements a partial library OS inside enclaves, excluding the storage, network, and threading functionalities [11]. SGX-LKL employs a complete library OS in enclaves, providing a minimal set of host interfaces [21]. Occlum, which is employed by OcclumJ, is also a library OS that supports secure and efficient multitasking inside enclaves [14].

For supporting Java confidential computing, some JVM-in-enclave solutions do exist. Civet is a framework for partitioning Java applications into enclaves [22]. It uses language-level defenses to harden the enclave interface, and also provides a partitioned JVM and garbage collection algorithm designed for enclaves. Uranus also partitions Java confidential applications [23]; it supplements Civet in that it provides a more efficient enclave boundary and a GC algorithm for data-intensive applications.

Comparatively, Lejacon removes JVMs from enclaves by performing AOT static compilation of the confidential code. As a result, Lejacon keeps applications' TCB sizes small and also allows NCC services to start/run quickly.

VII. Conclusion

Lejacon is a novel approach to Java confidential computing. It takes a balance between Java's platform independence and necessity of native execution in enclaves by a demand-driven programming paradigm, an SCW-directed separate compilation technique, and a runtime system for running confidential Java code. Lejacon is effective in protecting confidential Java applications, reducing their TCB sizes and in-enclave memory footprints at runtime.

VIII. Data Availability

The source code of Lejacon and use cases are publicly available.[2] The benchmarks used in the evaluation are also available.[3] Lejacon, along with its Java TEE SDK, becomes part of a universal secure computing platform.[4]

Acknowledgement

We would like to thank the researchers and engineers in the GraalVM community for their constructive feedback. Yuting Chen is the corresponding author. This research is supported by National Natural Science Foundation of China (Grant No. 62272296 and 62032004).

[2] https://github.com/apache/incubator-teaclave-java-tee-sdk
[3] https://github.com/MatthewXY01/Lejacon
[4] https://teaclave.apache.org/

References

[1] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," in *TrustCom/BigDataSE/ISPA 2015(1)*. IEEE, 2015.

[2] 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 *HASP@ISCA 13*. ACM, 2013.

[3] A. Rao, "Rising to the Challenge—Data Security with Intel Confidential Computing," 2022. [Online]. Available: https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141

[4] Microsoft, "Build with SGX enclaves - Azure Virtual Machines," 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/confidential-computing/confidential-computing-enclaves

[5] P. Karnati, "Data-in-Use Protection on IBM Cloud Using Intel SGX," 2019. [Online]. Available: https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx

[6] V. Costan and S. Devadas, "Intel SGX Explained," *IACR Cryptol. ePrint Arch.*, 2016. [Online]. Available: http://eprint.iacr.org/2016/086

[7] M. Schwarz, S. Weiser, and D. Gruss, "Practical Enclave Malware with Intel SGX," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019.

[8] R. Leslie-Hurd, D. Caspi, and M. Fernandez, "Verifying Linearizability of Intel® Software Guard Extensions," in *CAV 2015 (2)*, ser. LNCS, vol. 9207. Springer, 2015.

[9] Intel, "Understanding Intel® Software Guard Extensions (Intel® SGX)." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions-enhanced-data-protection.html

[10] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *OSDI 16*. USENIX Association, 2016.

[11] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *ATC 17*. USENIX Association, 2017.

[12] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux Applications With SGX Enclaves," in *NDSS 17*. The Internet Society, 2017.

[13] H. Tian, Y. Zhang, C. Xing, and S. Yan, "Sgxkernel: A Library Operating System Optimized for Intel SGX," in *Proceedings of the Computing Frontiers Conference*, 2017.

[14] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX," in *ASPLOS '20*. ACM, 2020.

[15] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic Application Partitioning for Intel SGX," in *USENIX ATC 17*. USENIX Association, 2017.

[16] A. Atamli-Reineh and A. P. Martin, "Securing Application with Software Partitioning: A Case Study Using SGX," in *SecureComm 15*, ser. LNICST, vol. 164. Springer, 2015.

[17] "CVE-2022-23302," 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23302

[18] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. R. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX," in *Middleware 16*. ACM, 2016.

[19] S. Brenner, T. Hundt, G. Mazzeo, and R. Kapitza, "Secure Cloud Micro Services Using Intel SGX," in *DAIS*, ser. LNCS, vol. 10320. Springer, 2017.

[20] L. Coppolino, S. D'Antonio, G. Mazzeo, and L. Romano, "A Comparative Analysis of Emerging Approaches for Securing Java Software with Intel SGX," *Future Generation Computer Systems*, vol. 97, 2019.

[21] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch, "SGX-LKL: securing the host OS interface for trusted execution," 2019. [Online]. Available: http://arxiv.org/abs/1908.11143

[22] C. che Tsai, J. Son, B. Jain, J. McAvey, R. A. Popa, and D. E. Porter, "Civet: An Efficient Java Partitioning Framework for Hardware Enclaves," in *USENIX Security 20*, 2020.

[23] J. Jiang, X. Chen, T. Li, C. Wang, T. Shen, S. Zhao, H. Cui, C.-L. Wang, and F. Zhang, "Uranus: Simple, Efficient SGX Programming and Its Applications," in *ASIA CCS '20*. ACM, 2020.

[24] "SOFABoot on Occlum," 2022. [Online]. Available: https://github.com/occlum/occlum/tree/master/demos/sofaboot

[25] Oracle, "HotSpot." [Online]. Available: https://openjdk.org/groups/hotspot/

[26] Eclipse, "OpenJ9." [Online]. Available: https://www.eclipse.org/openj9/

[27] S. C. Misra and V. C. Bhavsar, "Relationships Between Selected Software Measures and Latent Bug-Density: Guidelines for Improving Quality," in *ICCSA 03*, ser. LNCS, vol. 2667. Springer, 2003.

[28] B. C. Xing, M. Shanahan, and R. Leslie-Hurd, "Intel®Software Guard Extensions (Intel®SGX) Software Support for Dynamic Memory Allocation inside an Enclave," in *HASP@ISCA 16*. ACM, 2016.

[29] Intel, "Intel Software Guard Extensions Developer Guide," 2021. [Online]. Available: https://download.01.org/intel-sgx/sgx-linux/2.15/docs/Intel_SGX_Developer_Guide.pdf

[30] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, 1976.

[31] M. Mishra and J. Kar, "A Study on Diffie-Hellman Key Exchange Protocols," *International Journal of Pure and Apllied Mathematics*, vol. 114, 2017.

[32] 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*, vol. 13, no. 7. ACM, 2013.

[33] Oracle, "Java API Reference." [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html

[34] D. Bonetta, "GraalVM: Metaprogramming inside a Polyglot System," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*, 2018.

[35] L. C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz, "Comparing Points-to Static Analysis with Runtime Recorded Profiling Data," in *PPPJ'14*. ACM, 2014.

[36] B. Hardekopf and C. Lin, "The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code," in *PLDI 2007*, 2007.

[37] B. Liu and J. Huang, "SHARP: Fast Incremental Context-Sensitive Pointer Analysis for Java," in *OOPSLA 2022*. ACM, 2022.

[38] D. He, J. Lu, and J. Xue, "Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis," in *ECOOP 2022*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[39] Oracle, "GraalVM Native Image," 2021. [Online]. Available: https://www.graalvm.org/21.3/reference-manual/native-image/

[40] ——, "The Substrate VM Project," 2020. [Online]. Available: https://docs.oracle.com/en/graalvm/enterprise/20/docs/reference-manual/native-image/SubstrateVM/

[41] J. M. Piquer, "Indirect Mark and Sweep: A Distributed GC," in *International Workshop on Memory Management IWMM 95*, ser. LNCS, vol. 986. Springer, 1995.

[42] Y. Ossia, O. Ben-Yitzhak, and M. Segal, "Mostly Concurrent Compaction for Mark-Sweep GC," in *ISMM 04*. ACM, 2004.

[43] S. Richthofer, "Garbage Collection in JyNI - How to Bridge Mark/Sweep and Reference Counting GC," *CoRR*, vol. abs/1607.00825, 2016.

[44] "SGX Drive and SDK." [Online]. Available: https://download.01.org/intel-sgx/sgx-dcap/1.12/linux/distro/ubuntu18.04-server/

[45] Oracle, "OpenJDK 11," 2019. [Online]. Available: https://openjdk.org/projects/jdk/11/

[46] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski, "Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives," 2018.

[47] P. Yuhala, J. Ménétrey, P. Felber, V. Schiavoni, A. Tchana, G. Thomas, H. Guiroux, and J.-P. Lozi, "Montsalvat: Intel SGX Shielding for GraalVM Native Images," in *Middleware '21*. ACM, 2021.

[48] Legion of the Bouncy Castle Inc., "Bouncy Castle Java Distribution." [Online]. Available: https://github.com/bcgit/bc-java

[49] Alibaba, "Alibaba Druid." [Online]. Available: https://github.com/alibaba/druid

[50] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors," Cryptology ePrint Archive, Paper 2016/204, 2016, https://eprint.iacr.org/2016/204. [Online]. Available: https://eprint.iacr.org/2016/204

[51] O. Weisse, V. Bertacco, and T. Austin, "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves," in *ISCA '17*. ACM, 2017.

[52] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury, "Automated Partitioning of Android Applications for Trusted Execution Environments," in *ICSE '16*. ACM, 2016.
[53] Y. Wang, Y. Shen, C. Su, K. Cheng, Y. Yang, A. Faree, and Y. Liu, "CFHider: Control Flow Obfuscation with Intel SGX," in *INFOCOM 19*. IEEE, 2019.
[54] Y. Xiao, D. Park, A. Butt *et al.*, "Reducing FPGA Compile Time with Separate Compilation for FPGA Building Blocks," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019.
[55] J. Privat and R. Ducournau, "Link-Time Static Analysis for Efficient Separate Compilation of Object-Oriented Languages," in *PASTE '05*. ACM, 2005.
[56] C. Höger, F. Lorenzen, and P. Pepper, "Notes on the Separate Compilation of Modelica," in *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Linköping University Electronic Press, 2010.
[57] G. Fourtounis and N. S. Papaspyrou, "Supporting Separate Compilation in a Defunctionalizing Compiler," in *2nd Symposium on Languages, Applications and Technologies*, vol. 29. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.
[58] M. Murphy, J. Marathe, G. Bharambe, S. Lee, and V. Grover, "Separate Compilation in a Language-Integrated Heterogeneous Environment," in *Languages and Compilers for Parallel Computing - 26th International Workshop, LCPC 2013*, ser. LNCS, vol. 8664. Springer, 2013.
[59] B. Niu and G. Tan, "Monitor Integrity Protection with Space Efficiency and Separate Compilation," in *CCS '13*. ACM, 2013.
[60] Ali, Karim, "The Separate Compilation Assumption," PhD Thesis, UWSpace, 2014. [Online]. Available: http://hdl.handle.net/10012/8835
[61] ARM, "Building A Secure System using TrustZone Technology." [Online]. Available: https://developer.arm.com/documentation/PRD29-GENC-009492/c
[62] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Comput. Surv.*, vol. 51, no. 6, 2019.
[63] AMD, "AMD Memory Encryption." [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v9-Public.pdf
[64] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," in *USENIX Security 16*. USENIX Association, 2016.
[65] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable Memory Protection in the PENGLAI Enclave," in *OSDI 21*. USENIX Association, 2021.
[66] A. Baumann, M. Peinado, and G. C. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *OSDI 2014*. USENIX Association, 2014.