Uranus: Simple, Efficient SGX Programming and its Applications

Jianyu Jiang[†], Xusheng Chen[†], TszOn Li[†], Cheng Wang[†],
Tianxiang Shen[†], Shixiong Zhao[†], Heming Cui[†*], Cho-Li Wang[†], Fengwei Zhang[§]
{jyjiang,xschen,toli2,cwang2,txshen2,sxzhao,heming,clwang}@cs.hku.hk,zhangfw@sustech.edu.cn
[†]The University of Hong Kong [§]Southern University of Science and Technology

ABSTRACT

Applications written in Java have strengths to tackle diverse threats in public clouds, but these applications are still prone to privileged attacks when processing plaintext data. Intel SGX is powerful to tackle these attacks, and traditional SGX systems rewrite a Java application's sensitive functions, which process plaintext data, using C/C++ SGX API. Although this code-rewrite approach achieves good efficiency and a small TCB, it requires SGX expert knowledge and can be tedious and error-prone. To tackle the limitations of rewriting Java to C/C++, recent SGX systems propose a code-reuse approach, which runs a default JVM in an SGX enclave to execute the sensitive Java functions. However, both recent study and this paper find that running a default JVM in enclaves incurs two major vulnerabilities, Iago attacks, and control flow leakage of sensitive functions, due to the usage of OS features in JVM.

In this paper, Uranus creates easy-to-use Java programming abstractions for application developers to annotate sensitive functions, and Uranus automatically runs these functions in SGX at runtime. Uranus effectively tackles the two major vulnerabilities in the code-reuse approach by presenting two new protocols: 1) a Java bytecode attestation protocol for dynamically loaded functions; and 2) an OS-decoupled, efficient GC protocol optimized for data-handling applications running in enclaves. We implemented Uranus in Linux and applied it to two diverse data-handling applications: Spark and ZooKeeper. Evaluation shows that: 1) Uranus achieves the same security guarantees as two relevant SGX systems for these two applications with only a few annotations; 2) Uranus has reasonable performance overhead compared to the native, insecure applications; and 3) Uranus defends against privileged attacks. Uranus source code and evaluation results are released on https://github.com/hku-systems/uranus.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

SGX; Java; JVM; Iago Attack; TEE; Side-channel; Type-safety; Spark; ZooKeeper; Data-handling; Big-data; Garbage Collector (GC);

*Corresponding author.

ASIA CCS '20, October 5–9, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-6750-9/20/06.
https://doi.org/10.1145/3320269.3384763

ACM Reference Format:
Jianyu Jiang[†], Xusheng Chen[†], TszOn Li[†], Cheng Wang[†], and Tianxiang Shen[†], Shixiong Zhao[†], Heming Cui[†*], Cho-Li Wang[†], Fengwei Zhang[§]. 2020. Uranus: Simple, Efficient SGX Programming and its Applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3320269.3384763

1 INTRODUCTION

The cloud computing paradigm enables various data-handling applications (e.g., Spark [76]) to be deployed in public clouds. Since these applications inherently desire reliability and security, and Java has diverse security features (e.g., type-safety), many of these applications are written in Java. However, the JVM runtime alone is insufficient to defend against privileged attacks, because adversaries may control the entire cloud software stack, including OS kernels [59, 77].

Recently, Intel Software Guard eXtensions (SGX) [38] becomes a promising technique on protecting sensitive data for data-handling applications in public clouds. SGX provides an *enclave* execution abstraction with limited memory (typically, about 100MB). SGX runs trusted code in an enclave and uses the CPU hardware to prevent attackers from seeing or tampering with the code and data in the enclave. To allow untrusted code to invoke functions in an enclave, SGX provides the ECall API in C/C++.

Traditionally, SGX systems for Java applications (e.g., SecureKeeper [28] and Opaque [77]) adopt a code-rewrite approach. In this approach, the developers of an application partition Java code into the trusted part, including all sensitive functions which process plaintext data, and the untrusted part. They completely rewrite the trusted code from Java to C/C++ using SGX API and Java Native Interface (JNI), and then run the rewritten code in enclaves. This approach can maintain a minimum TCB and memory footprint because it runs only the rewritten code in enclaves. However, this code-rewrite approach often requires non-trivial efforts from developers, including rewriting all dependent Java libraries into C/C++ code, which could be tedious and error-prone. Worse, the rewritten code loses the type-safety of Java.

To tackle the limitations caused by rewriting Java code, recent systems (e.g., SGX-Spark [11] and CordaSGX [4]) take a code-reuse approach. In this approach, a JVM is run within an enclave to execute the trusted Java code using a dedicated thread, and encrypted data is passed into the enclave for the thread to decrypt and to compute on. For instance, SGX-Spark runs an unmodified JVM using SGX-LKL [10] to execute only the Spark user-defined-functions (UDF) in enclaves, and the other parts of Spark are run out of enclaves. Running a dedicated JVM in an enclave greatly eases the

deployment of trusted Java code and preserves type-safety, so this code-reuse approach becomes increasingly popular.

However, despite much effort on developing advanced SGX systems [4, 11, 54] by this code-reuse approach, two major challenges remain in these systems. First, running a default JVM in an enclave can easily incur severe attack surface, which can expose or change the control flow of the trusted Java code. Specifically, JVM uses OS features frequently at runtime for efficiency, and attackers out of enclaves can infer the control flow of the trusted code by observing OS events. For example, our study (§4.4) found out that an attacker can abuse its control of system signals to get the size of an object in an enclave memory allocation, which reveals the control flow and even the plaintext data in enclaves. Moreover, applications running within enclaves may be vulnerable to Iago attacks [29], and recent work [68] shows that such attacks still widely exist during enclave transitions in SGX systems. Our study (§4) confirmed that Iago attacks are more pronounced when porting the default JVM into enclaves, since doing so results in many enclave transitions.

The second challenge is that a JVM running in an enclave can incur severe performance degradation for data-intensive applications. Specifically, JVM's default GC reclaims objects when there is not enough space for allocating a new object. Since the SGX memory is merely around 100MB, the GC needs to frequently stop all threads to scan the entire heap shared by multiple threads. After all, JVM's default GC is designed to manage GBs of memory and lacks an efficient mechanism to reclaim memory for enclaves.

We present Uranus[1], the first SGX system to tackle these two challenges and to efficiently protect Java applications. Uranus provides two high-level Java programming abstractions: `JECall` and `JOCall`. An application developer can use `JECall` to annotate functions in trusted code, and such annotated but unmodified functions and their callees will be executed in an enclave automatically. If a function is annotated with `JOCall` and its caller is running in an enclave, this annotated function will be executed outside.

To completely tackle the first challenge while maintaining a small TCB, Uranus includes four JVM components (i.e., GC, dynamic code loader, JIT and exception handler) in an enclave. Our methodology to eliminate their attack surface is isolating these components from outside enclaves and verifying all content passed into enclaves. A key novel component in Uranus is an OS-decoupled, thread-safe GC protocol. This protocol is developed on one observation: Uranus's JVM runtime contains sufficient application bytecode structures (e.g., basic block back-edges and function entry points) for doing code instrumentation, so threads in an enclave can be efficiently stopped without going across the enclave boundary. Therefore, Uranus's GC protocol eliminates transitions across the enclave boundary, effectively protecting the confidentiality and integrity of the control flow of the trusted Java code. Overall, Uranus's GC is completely isolated from outside enclaves, including OS, so the first challenge is tackled in GC.

Uranus also tackles the first challenges in the other three JVM components. For the dynamic code loader, we design and implement a class-level bytecode attestation protocol, which effectively verifies the integrity of the bytecode loaded into in an enclave at runtime and hides the control flow of the bytecode. For the JIT compiler, we leverage the bytecode-to-assembly template in OpenJDK's interpreter to build an simple and efficient JIT with full support of all Java-8 bytecode instructions, while maintaining a small TCB. This JIT is completely isolated from outside enclaves; it prevents Iago attacks during enclave transitions by conducting sanity checks on the parameters passed through `JECall`/`JOCall`. Uranus's exception handler also runs entirely within an enclave without involving any OS feature.

Our observation to tackle the second challenge is that, although a thread in a data-handling application often allocates many objects within an enclave, only few objects are shared among threads. Therefore, unshared objects can be efficiently reclaimed whenever a thread finishes a `JECall`. With this observation, Uranus introduces a region-based enclave memory management technique, which mostly avoids stopping all threads in an enclave and efficiently reclaims per-thread objects whenever a thread finishes a `JECall`.

We implemented Uranus in OpenJDK on Linux. In SGX practice, the trusted code may read data from untrusted memory outside enclaves, which may infect the control flow of the trusted code and compromise the integrity of its computation result. Leveraging the type-safety of Java, Uranus includes a runtime checking protocol to prevent the bytecode from running in an enclave accessing memory outside (§4.3). Therefore, even if application developers omit to annotate some sensitive functions, this protocol prevents the trusted code from running in enclaves leaking plaintext data to these functions running outside. Overall, Uranus achieves a small TCB: all Uranus components running in enclaves, including the four OS-decoupled JVM components, have only 25.2k LoC.

We integrated Uranus with two data-handling applications written in Scala and Java: Spark [76] and ZooKeeper [36]. Spark-Uranus achieves the same confidentiality and integrity guarantees as Opaque's encryption mode [77]; ZooKeeper-Uranus achieves the same security guarantees as SecureKeeper [28]. We compared Spark-Uranus to Opaque [77] (encryption mode) and ZooKeeper-Uranus to SecureKeeper [28]. For Spark-Uranus, we included all 8 big-data queries evaluated in Opaque [77]. Evaluation shows that:

- Uranus is easy to use. We annotated only two or four functions for each application. Spark-Uranus runs unmodified UDF queries in enclaves.
- Uranus is efficient. For ZooKeeper-Uranus, it incurred merely up to 19.4% performance overhead compared to the native (insecure) executions. Spark-Uranus incurred 1.2X to 7.6X performance overhead compared to native Spark on typical dataset sizes. Partly due to Uranus's new GC protocol, Spark-Uranus is the first SGX work that supports typical big-data dataset sizes [76], two to three orders of magnitude larger than the dataset sizes evaluated in Opaque.
- Uranus effectively tackled privileged attacks.

The main novelty of this paper is two new protocols: 1) an OS-decoupled, thread-safe GC protocol that enables Java big-data applications to run efficiently on the limited enclave memory; and 2) a first integrity attestation protocol for dynamically loaded Java bytecode. Uranus's GC protocol can be integrated in existing SGX big-data systems (e.g., SGX-Spark), greatly improving enclave memory efficiency and reducing attack surface in these systems.

[1]Uranus, an ancient Greek god, brings order and safety to the cosmic chaos.

The remaining of the paper is organized as follows. §2 introduces SGX and JVM background. §3 gives an overview of the Uranus framework. §4 introduces Uranus's runtime. §5 gives the implementation details. §6 shows our evaluation results. §7 presents related work and §8 concludes.

2 BACKGROUND

2.1 Intel SGX

An SGX enclave isolates the execution environment of the application code and data running inside and protects them from outside privileged access, including OS, hypervisor, and BIOS. Memory pages belonging to an enclave reside in the *Enclave Page Cache* (EPC). EPC has a total size of 128MB per CPU, and only around 100MB can be used by application code. If the code running in the enclave uses more than 100MB, a slow SGX paging mechanism will incur a 1,000X slowdown compared to regular OS paging [28]. An SGX enclave can execute only user-space instructions, so the enclave code has to do OCalls to leave enclaves for system calls. When an interrupt or hardware exception (e.g., General Fault) is raised in an enclave, the processor performs an Asynchronous Enclave Exit (AEX) to handle the exception or interrupt. Though AEX and OCall do not directly leak secrets in enclaves, they can result in severe side-channels or even attack surface (§4.4) for revealing the control flow of trusted code and plaintext data.

2.2 Java Virtual Machine (JVM)

We denote Hotspot [6], the most popular implementation of Java Virtual Machine, as JVM. For portability and language features (e.g., Reflection), JVM loads Java bytecode at runtime and executes it with either an interpreter or JIT compiler. JVM runs the interpreter initially, if some bytecode is executed frequently (known as hotspot), JVM uses JIT to compile this code into highly optimized machine code to speed up the execution. Compared to the interpreter (16k LoC), verifying the correctness of the JIT compiler is much more difficult because JIT is much larger (210k LoC) and contains complex optimization logic. Therefore, we built a simple JIT for Uranus based on the bytecode-to-assembly template in the default JVM interpreter (§4.2). JVM has about 1 million LoC, and will result in a large TCB when running in an enclave. When the code of a type-safe language (e.g., Java) is executed in a bug-free JVM, JVM's runtime checks ensure that the code is memory-safe and has no memory leaks or buffer overflow bugs.

3 OVERVIEW

3.1 Threat Model

Uranus is designed for applications running in a client-server manner, and its threat model is the same as typical SGX systems (e.g., VC3 [59] and SecureKeeper [28]). Specifically, SGX, clients, all Uranus's components running in clients and enclaves, and a server application's functions running in enclaves are trusted.

Other hardware and software layers such as BIOS, hypervisor, OS, Uranus's components outside enclaves, and server application code running outside enclaves can all be controlled by attackers and therefore are untrusted. Attackers can access and tamper with memory, observe and hijack system calls, and drop network packages.

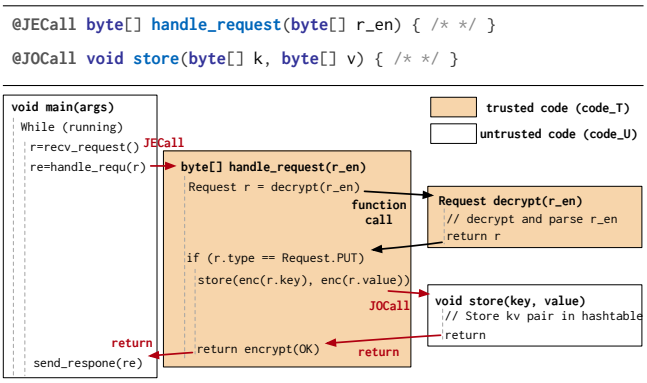

Figure 1: Code, `JECall`/`JOCall` annotations and workflow of the key-value store running on Uranus. `enc` encrypts the key and value.

Same as SGX-Spark and VC3, denial of Service (DoS) and micro-architectural side-channel attacks (e.g., CPU cache side-channels) are out of the scope of this paper. Nevertheless, Uranus's threat model considers two types of software-level side-channels, AEX and OCall, caused by running a default JVM in an enclave. For instance, when a JVM's default GC is invoked, all threads within the enclave will trigger AEX via OS signals and leave the enclave, leading to control flow leakage.

3.2 Uranus's Programming Method

To preserve confidentiality and integrity for server applications, developers partition code into two parts: trusted code ($\texttt{Code}_\texttt{T}$) running in enclaves and untrusted code ($\texttt{Code}_\texttt{U}$) running outside enclaves. Uranus has two annotations `JECall` and `JOCall` for applications to run $\texttt{Code}_\texttt{T}$'s functions in enclaves. Functions annotated with `JECall` and their callees are in $\texttt{Code}_\texttt{T}$. When a function annotated with `JECall` is invoked, the application execution transits into an enclave until the function returns. When a function annotated with `JOCall` is invoked in $\texttt{Code}_\texttt{T}$, the application execution transits out of the enclave until the function returns.

Figure 1 shows a key-value server program, simplified from ZooKeeper [13]; the partition of this program is based on the partition from SecureKeeper [28]. In the program, `handle_request` and its callees (e.g., `decrypt`) are in $\texttt{Code}_\texttt{T}$, while the other functions are in $\texttt{Code}_\texttt{U}$. Initially, the program runs outside enclaves, and Uranus creates one enclave for it. When the program receives an encrypted client request `r_en`, it calls `handle_request` to decrypt `r_en` to `r` and conducts a `get` or `put` operation. Since `handle_request` is annotated as `JECall`, the execution transits into an enclave. All functions called by `handle_request` (e.g., `decrypt`) run in enclaves, except the `store` function annotated as `JOCall`. `store` stores encrypted key-value pairs outside enclaves, maintaining a small TCB and low enclave memory footprint.

3.3 Architecture

Figure 2 shows Uranus's architecture. Uranus consists of five trusted components: a code integrity verifier, an JIT compiler, an enclave adaptor, a garbage collector, and an exception handler. For a server application, Uranus creates one enclave in the local machine and loads these components into the enclave.

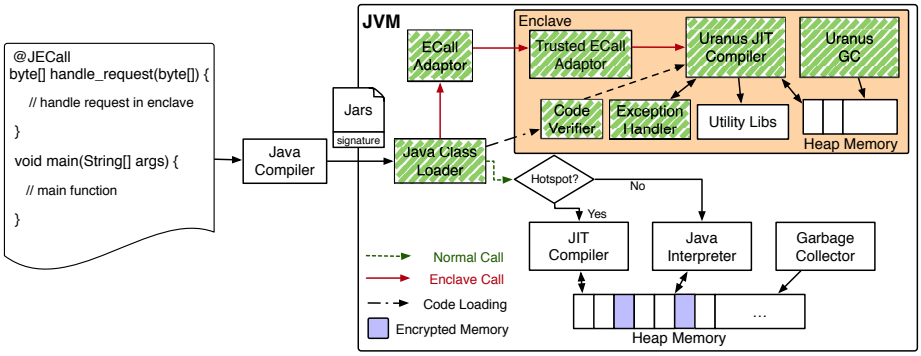

Figure 2: The architecture of URANUS. URANUS's components are shaded.

Code Verifier (§4.1) loads only the classes executed in the enclave and verifies the hashes of the classes for their integrity, maintaining a small TCB. **URANUS's JIT Compiler** (§4.2) executes Java bytecode in an enclave. It also handles `JECall`/`JOCall` transitions with adaptors. Utility libraries (e.g., math) are included in the enclave to provide necessary functionalities. **Garbage Collector** (§4.4) manages the enclave's heap memory efficiently.

These components do not incur extra threats for confidentiality and integrity compared with the low-level SGX programming model. For confidentiality, URANUS's communication channels (i.e., Code Verifier and ECall Adaptor) between an enclave and untrusted world do not leak secret data (§4). For integrity, URANUS provides a protocol (§4.1) for clients to attest the integrity of the code dynamically loaded into server applications' enclaves. URANUS's GC, JIT and Exception Handler run entirely in enclaves do not incur additional threats as they do not communicate with outside.

4 URANUS RUNTIME

The existing SGX programming approach in C/C++ takes a static way to guarantee code integrity [41]. It generates a digest using the initial state of all server applications' statically compiled code that may be used in enclaves and exchanges the digest with clients for attesting the integrity of the code (i.e., attestation).

However, this static compilation approach is unsuitable for languages (e.g., Java and JavaScript) that run code in a dynamically loading manner, because the dynamically loaded code is not included in the initial state. To fix this issue, SecureWorker [9] statically compiles the JavaScript language runtime with application code and all its dependent libraries, leading to a huge TCB. For instance, the SpiderMonkey libraries are 521k LoC [12]. Moreover, this approach cannot support dynamically loading code determined at runtime (e.g., through Java Reflection).

A key requirement for URANUS's code loading protocol is that it must not expose attack surface to attackers outside an enclave. For instance, Java's default code loading method is to load each class on demand (when it is needed for executions), but doing so in an enclave will require loading each class from outside the enclave, which exposes control flow to attackers.

URANUS's code loading protocol is inspired by a static loading protocol of dependent libraries, developed in Graphene-SGX [67]. Graphene-SGX loads executable and a manifest containing hashes of dependent dynamic libraries during an enclave initialization, and computes a hash of the executable and the manifest for clients to verify. URANUS extends this static protocol in two aspects, leading to a new runtime loading protocol. First, URANUS loads code in the class level instead of the library level in order to minimize the code size loaded into enclaves. Our analysis of one of our applications, Spark-URANUS, shows that loading only `JECall`'s dependent classes consumes over 90% less enclave memory than loading `JECall`'s dependent libraries (Appendix A). Second, URANUS supports loading classes provided at runtime (e.g., Spark UDF).

Given a version of application code, developers first compute SHA256 hashes for all classes in a jar using `jarsigner` in OpenJDK. Then, they use `URANUS-dep` provided by URANUS to automatically find all dependent classes of all `JECall` (for the integrity of enclave code, in §4.1) and all static invocation sequences of these `JECall` (for the execution integrity of a sequence of `JECall` invocations, in §4.3). Note that `URANUS-dep`'s class dependency is deterministic: each version of application code has only one dependency. It also generates a manifest of all `JECall`, `JOCall`, the static `JECall` invocation sequences and package version. During the initialization of an application enclave, URANUS loads all dependent classes into the enclave and generates a measurement (hash) of both these classes' hash and the manifest. This measurement serves as proof of the integrity of loaded Java bytecode.

4.1 Integrity of Code Loading

Figure 3 shows the workflow of this protocol with six steps. **(1)** URANUS creates an enclave using standard SGX API for each server application. The enclave contains all URANUS's trusted components but does not contain any application code. **(2)** When a server application launches, the (untrusted) URANUS loader computes the dependent classes of all JECalls and invokes the trusted URANUS loader in the enclave to copy all these classes' content, their hashes and the manifest into the enclave. **(3)** URANUS's trusted loader first verifies each class's content with its hash, and then computes a hash H_0' from both all classes' hash and the manifest file. Note that the content of these classes is not parsed at this moment.

(4) When a remote client (trusted) tries to connect to the server, the client attests URANUS's trusted components following SGX's standard attestation protocol and establishes a secure communication channel using Diffie–Hellman [31]. **(5)** The enclave reveals H_0' to the client. The client computes an H_0 locally using `URANUS-dep` for each version of an application's code and caches H_0. The client compares H_0' with H_0, rejects the enclave if they are different. Note that this key-exchange process only happens once when the client

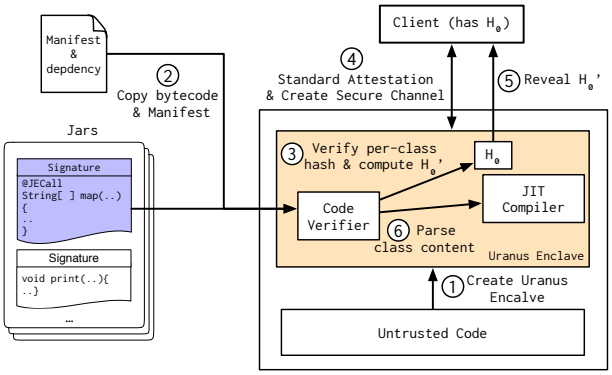

Figure 3: Uranus's Attestation Protocol.

connects. **(6)** When a function (`JECall` or its callees) is called for the first time, Uranus parses the content of the function's class to Uranus's JIT compiler for executions.

In addition to `JECall` and dependent classes, Uranus also preserves integrity of dynamically loaded code using Reflection (e.g., Spark's UDF). Uranus provides an API (i.e., `loadClass (className)`) for the trusted code to load a class `C` and its dependent classes into an enclave. This API runs step **(2)** ~ **(3)** of the code loading protocol, except that class dependencies are computed at runtime using `Uranus-dep` (§5). `loadClass` returns a hash H_c' of all the classes' hashes, and H_c' is compared with a client provided H_c for integrity. Since `Uranus-dep` is ran outside the enclave, an attacker can forge the dependencies, but clients can detect this attack by comparing H_c' and H_c.

Security Analysis: Uranus guarantees the integrity of bytecode being loaded and runs within an enclave, as all `JECalls`' dependent classes' hashes are kept in an enclave. If an attacker alters the content of a class before it is loaded, Uranus's verification of class content (Step 3) fails, and the enclave exits. When the attacker provides a fake Java class and its corresponding hash, or incorrect dependent classes to the enclave, clients will reject the enclave (Step 5) as H_0' is different from H_0.

4.2 Executing Java Bytecode in Enclaves

Java bytecode can be executed by an interpreter or a JIT compiler. The interpreter in Hotspot (JVM) fetches a bytecode before jumping to its assembly code snippet, generated by the interpreter. This fetch-and-dispatch process adds many branch instructions to the assembly code and often causes time-consuming missed branch predictions (§6.3). The JIT compiler in Hotspot is much faster than the interpreter because it translates multiple Java bytecode to assembly code, greatly reducing the branches of fetch-and-dispatch. This JIT also contains an intermediate representation (IR) process with complex optimizations during the translation to improve performance. The IR and its optimizations have a large TCB (210k LoC).

To maintain a small TCB, we first built a simple interpreter based on OpenJDK's interpreter [6]. Unfortunately, the interpreter was too slow for diverse workload with loops (§6.3). To improve performance, we built a simple JIT compiler for Uranus based on Hotspot's interpreter. Uranus's JIT directly compiles each Java method's bytecode to assembly code using the interpreter's bytecode-to-assembly template. Uranus's JIT excludes the IR and its optimizations, because they contain a large TCB and are hard to verify [44]. Appendix B shows the assembly code generated by the JIT.

Uranus's enclave adaptor handles transitions of `JECall`/`JOCall`. In such a transition, an Iago attack may incur: an attacker can manipulate an object reference passed to `JECall`'s parameters, and make the reference point to an arbitrary memory address within the enclave. If such manipulations are not detected, the trusted code within enclaves may write to this arbitrary memory address, easily leading to various attacks (e.g., ROP attacks [24, 48]).

To eliminate Iago attacks from breaking type-safety of enclave code during enclave transitions, when a thread calls a function annotated as `JECall`, Uranus's adaptor copies stack elements into an enclave, verifies if it is a valid `JECall` by checking the function's metadata in the manifest, parses the corresponding loaded class file and compiles the code (if the class has not been parsed in step **(6)** in Figure 3). Then, the thread starts executing the `JECall` within the enclave. Since stack elements are copied from outside, the adaptor scans the stack content to ensure that no object reference in the stack points to enclave memory. Uranus's adaptor does the same check for the returned value of `JOCall`.

Security Analysis: Uranus's JIT does not incur any new attack surface as it is completely isolated from outside the enclave. It takes only verified class files from the dynamic code loader and compiles them to native code, which does not require help from OS or leaving enclaves. Therefore, attackers outside enclaves cannot infer the execution flow of the trusted bytecode by observing OS signals or AEX. Uranus's enclave transitions eliminate Iago attacks. Overall, Uranus's JIT compiler supports all 203 bytecode instructions of Java, achieves good performance on applications (§6.3), yet adds only 2.1k LoC to the interpreter's TCB.

4.3 Ensuring Enclave Confidentiality and Integrity at Runtime

This paper requires the developers of an application to make efforts to partition the trusted code and untrusted code. Developers should have sufficient knowledge to include all or most sensitive functions in the trusted code partition, or they can use static analysis tools (e.g., Glamdring [51], Civet [16]) to infer the trusted partition. Developers then use Uranus's `JECall` and `JOCall` annotations to realize the trusted and untrusted code partition. Specifically, they add decryption functions to decrypt data passed into the entry points of the partition, and add encryption functions to encrypt computation results or updated data passed through the exit points of the partition. This paper assumes that the entry and exit points of the partition are correctly identified by developers, which is also required by Glamdring.

To prevent code running within enclave from leaking secret within enclaves to outside, Uranus enforces a tight boundary between trusted bytecode and untrusted bytecode at runtime by leveraging the type-safety of Java. Specifically, Uranus forbids enclave code from accessing untrusted memory, unless these accesses use Uranus's untrust-memory-access API. To ease discussion, $\texttt{Code}_\texttt{T}$ denotes trusted application code running in an enclave. $\texttt{O}_\texttt{T}$ denotes

Algorithm 1: SafeGetfield(obj, field_name, type)

```
1 offset = field_metadata(obj, field_name);
2 if (obj + offset) ∈ [Enclave_start, Enclave_end) then
3 |   abort;
4 else
5 |   val = *(obj + offset);
6 |   if type == Object and val ∈ [Enclave_start, Enclave_end) then
7 |   |   abort;
8 |   else
9 |   |   return val;
```

objects located in an enclave's heap, and O_U denotes objects located in the outside heap.

This boundary is enforced by two properties using runtime checks injected in compiled code generated by Uranus's JIT. The first property is **read-integrity**: $Code_T$ (trusted code) does not read fields from O_U (objects created by untrusted code). This prevents $Code_T$ reading a value from outside enclaves and changing $Code_T$'s control flow.

The second property is **write-confidentiality**: $Code_T$ does not write any data in enclave memory to fields of O_U. When the data is written to untrusted memory, it is possible that the value is computed from sensitive input data and thus also sensitive. Writing this value to untrusted memory will probably break enclave confidentiality. By enforcing this property, even if developers carelessly omit to annotate some sensitive functions, the bytecode running inside enclaves is forbidden to write any data to these functions running outside enclaves via memory access.

Read-integrity and write-confidentiality are enforced by simply checking if a bytecode accesses objects in an enclave. These checks compare an object pointer with the constant bound of enclave memory (i.e., $[Enclave_{start}, Enclave_{end})$), throwing an error if the object is out of bound . The bound is cached in two global variables and located in CPU registers for efficiency. As JVM provides an `Unsafe` API to access raw memory of an object, Uranus also includes bound checks in `Unsafe`. Uranus also prevents trusted code invoking O_U's member functions in enclaves, as JVM decides the entry point of an object's member function according to its class at runtime due to polymorphism. If such function calls are allowed, attackers outside enclaves can tamper with O_U's classes to inject arbitrary code.

In practice, some accesses to untrusted memory (e.g., reading encrypted data from outside enclave) are intended by developers and should be allowed. In such cases, developers can use Uranus's four untrust-memory-access API: `SafePutfield(obj, fieldname, val)`, `SafeGetfield(obj, fieldname, type)` and `SafeArrayCopy(src, dest, len)`. This API disables runtime checks of read-integrity and write-confidentiality as in the compiled enclave code.

Similar to enclave transitions, the untrusted memory accessed by this API must not contain a reference to any memory address within an enclave to avoid Iago attacks. Uranus does sanity checks on these objects references. Algorithm 1 shows the pseudo-code of `SafeGetfield` in Uranus, and `SafeGetfield` ensures the object references returned are not pointing to enclave memory. The checks in `SafePutfield` are similar to `SafeGetfield`'s, except that `SafePutfield` uses `val`'s type in line 6 instead of the one provided by developers in `SafeGetfield`. `SafeArrayCopy` ensures

```
1  @JEcall
2  public void udf_process(char[] secrets) {
3    char[] plaintext = decrypt(secrets);
4    Record record = deserialize(plaintext);
5    if (record.disease == "cancer") {
6      // CanerCheck is 10KB and incurs GC
7      ret = new CancerCheck(record).analyze();
8      // an attacker can observe GC and
9      // infer the disease is cancer
10   }
11 }
```

Figure 4: An Attack on OS-assisted GC.

that primitive array references (`src` or `dest`) pointing to enclave memory are in-bound. In fact, SGX programming in C/C++ faces a similar problem, as it requires developers to write code to check all cross-boundary memory access [38], tedious and error-prone.

Uranus's tight boundary ensures the execution integrity of each `JECall` invocation. However, an attacker can manipulate the invocation order of `JECall` to tamper with an application's execution integrity. To prevent this attack, Uranus uses the approach from Glamdring [51]: During each `JECall`, Uranus checks within enclave if the actual invocation sequence complies with one of the sequences computed by `Uranus-dep`, and aborts the enclave with an exception if not.

Security Analysis: Uranus provides stronger isolation and type-safety compared with the traditional SGX programming in C/C++. First, read-integrity and write-confidentiality enforce complete isolation between enclave objects and outside. Second, JVM's built-in checks in Uranus's JIT guarantees the type-safety of enclave objects. Third, runtime checks in `SafeGetfield` reject any returned object's reference pointing to enclave memory.

4.4 Memory Management (GC)

A naive approach for Uranus's GC is to directly adopt Hotspot's OS-assisted garbage collection. Hotspot stops all threads (i.e., Stop-The-World, or STW) during each GC, because some objects may be concurrently used by the GC thread and other threads. Hotspot makes use of OS signals (i.e., `segfault`) for STW. Specifically, Hotspot injects a memory read instruction on a special page before each back-edge of basic blocks, and this page is set unaccessible using `mprotect` by the GC thread. Therefore, all threads executing Java bytecode will incur `segfault` and be stopped. After all executing threads are stopped, the GC thread does a GC and then resumes the stopped threads.

However, simply adopting the approach is neither secure nor efficient in Uranus due to two issues. First, handling `segfault` requires the threads to leave an enclave and go through OS, which exposes significant attack surface. Figure 4 shows an example of how the OS-assisted GC leaks sensitive secrets of an enclave. An attacker can manipulate enclave heap [63] by frequently invoking the `JECall` such that enclave heap does not have enough space for a `CancerCheck` object. Therefore, the attacker can infer that a GC is invoked by observing the `segfault` signal in the OS level. The attacker can infer that the enclave has likely executed line 7, and that the encrypted record has cancer. Second, since an enclave has a small memory space, sharing one enclave heap among multiple threads incurs frequent GC and enclave transitions, greatly downgrading application performance.

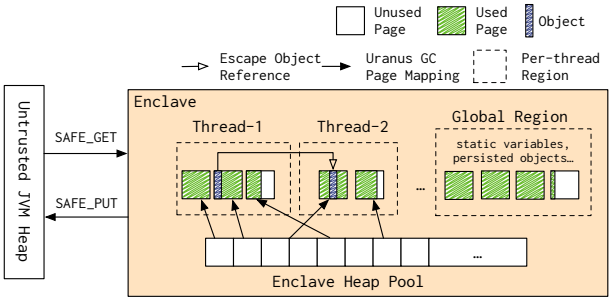

Figure 5: Regional Heap Architecture.

To tackle these two issues, Uranus presents an OS-decoupled, thread-safe, and efficient GC protocol. Figure 5 shows the architecture of Uranus's region-based GC. When a thread enters an enclave, Uranus allocates a region for the thread. The thread-private region consists of multiple large pages mapped from a global page pool. Each page is 32KB (adopted from the setting of Yak [55], a big-data friendly GC) instead of 4KB to reduce fragmentation. Each object has a `regionID` stored in its JVM object header. There is also a global heap that stores persisted objects across JECalls (e.g., static objects). Some objects in one region may escape to other regions. For example, if one `CancerCheck` object is written to a static object's field, then this object may be read by other threads and escape. Uranus captures an escape object while an object is written to a field of an object in different regions. `onObjectPutfield` in Algorithm 2 capturing escaped objects of a region. When two objects' region IDs are different, a mapping is added to `escapeMap` of the current thread (line 30), and this `onObjectPutfield` code is injected by Uranus's JIT to an enclave's compiled code. Note that `escapeMap` can be either appended or cleared, but is not truncated. This is because an object may transitively escape to multiple regions, and `escapeMap` denotes a history of escaped objects for each thread's region. Since Uranus need only track escaped objects, not Java primitives, `onObjectPutfield` incurred only 0.8% performance overhead for the big-data queries in our evaluation.

Algorithm 2 shows Uranus's complete GC protocol. When a thread finishes a `JECall` execution, a Uranus GC (`doGC`) is invoked. Doing a GC contains a fast path and a slow path. The fast path (line 12 ~ 13) simply clears all pages in the current thread's region, if the thread's `escapeMap` is empty.

The other lines in the `doGC` function show the slow path. The slow path first tries to invoke an OS-decoupled STW (line 15 ~ 27). If a thread `T` succeeds (`needGC` is atomically set `True` in line 20), it calls `startGC` to find all escaped objects by scanning all threads' stack and `escapeMap`, to migrate the escaped objects to the global region, and to clear all pages in thread `T`'s region. Thread `T`'s `escapeMap` entries, whose escaped objects' regionID differs from `T`'s regionID, will be migrated to the objects' corresponding regions. If `T` fails (i.e., another thread successfully sets `needGC`), `T` waits for the successful thread to finish and then retries.

When a thread cannot find enough memory for a new object in the enclave heap during execution, the thread also invokes a `doGC(False)` to conduct a standard MarkSweepCompact GC [6].

To implement STW, Uranus's JIT injects `onGCCheck` to all back-edges and function entries (including `JECalls`) in an enclave's compiled code. Importantly, for efficiency, line 7 does not involve

Algorithm 2: GC Protocol

```
Variables:
  bool needGC        // GC is being invoked
  int nEncGCThds     // # of threads stopped for a GC
  int nEncThds       // # of threads in an enclave
  Thread self        // Current thread, thread-private
1  Function onJECallEnter()
2      atomic_inc(nEncThds);
3  Function onJECallExit()
4      doGC(True);
5      atomic_dec(nEncThds);
6  Function onGCCheck()
7      while needGC do
8          atomic_inc(nEncGCThds);
9          while(needGC);
10         atomic_dec(nEncGCThds);
11 Function doGC(bool onExit)
12     if onExit and self.escapeMap is ∅ then
13         self.resetRegion() // Simply clear page mapping.
14     else
15         while True do
16             curGC = needGC;
17             if curGC then
18                 onGCCheck();
19                 continue;
20             gcFlag = compare_swap(&needGC, curGC, 1);
21             if !gcFlag then
22                 atomic_inc(nEncGCThds);
23                 while(nEncGCThds < nEncThds);
24                 self.startGC();
25                 atomic_dec(nEncGCThds);
26                 needGC = False;
27                 break;
28 Function onObjectPutfield(destObj, offset, obj)
29     if obj.regionID != val.regionID then
30         self.escapeMap ← (obj, (destObj, offset));
31     *(destObj + offset) = obj // Default putfield logic
```

an atomic operation, and a thread can finally be stopped at following back-edges or function entries. Moreover, because the frequency of GC is often low (`needGC` is `False` most of the time), checking the `needGC` flag incurs negligible overhead in our evaluation. On the start of a `JOCall`, Uranus simply decreases `nEncThds` atomically, but Uranus does not invoke a GC at this point because enclave memory needs to be used on the `JOCall`'s return. On the return of the `JOCall`, Uranus increases `nEncThds` atomically.

Security Analysis: Uranus's GC does not expose attack surface to attackers. Since thread synchronization is implemented without any OS help or leaving an enclave, an attacker cannot infer the invocation of a GC using OS signals, so Uranus's GC does not reveal trusted code's control flow. Moreover, Uranus's GC is type-safe. Algorithm 2's fast path can safely clear all objects in a thread's region when the thread finishes a `JECall`, because the thread's empty `escapeMap` means that no object has ever escaped during the `JECall` execution. Algorithm 2's slow path can find all escape objects, by scanning all threads' stack and `escapeMap`, as `escapeMap` is a history of objects escaped to other regions.

5 IMPLEMENTATION DETAILS

Uranus's implementation is based on OpenJDK-8, a popular open-source JVM. Uranus's runtime supports multiple utility libraries, including Java reflection API (e.g., `java.lang` and `Arrays`). Uranus's JVM codebase running in an enclave is merely 25.2k LoC, as it excludes `javax`, `java.security`, debugging, etc. A comparison of LoC between Uranus and other code-reuse systems is given in Appendix E. Uranus also supports necessary system calls such as standard I/O, Time and File. Uranus does a system call by simply calling an OCall that does the system call outside enclaves, similar to Panoply [62] which supports system calls with a minimized TCB. Uranus currently does not support thread operations (e.g., thread creation or thread-local storage) in the `java.lang.Thread` package. Multi-threading and synchronization on enclave objects are supported, while synchronizations across enclave boundaries fail with an `EnclaveException` exception. Details of implementing OS-decoupled multi-threading, synchronization and exception handling are given in Appendix C. Uranus's current implementation is sufficient to run all applications in our evaluation.

`Uranus-dep` is implemented using the Java ASM package [5], an easy-to-use library for analyzing Java bytecode file. To obtain class dependencies for `JECall` (§4.1), `Uranus-dep` begins by traversing a class file (`JECall`'s) and finding out dependent classes of the current class, then puts the dependencies into an `AdjacencyList`. This is executed recursively until all classes are found. To obtain the invocation sequences of `JECall` (§4.3), `Uranus-dep` traverses all functions invoked by `main` recursively and constructs a set of static invocation sequences of `JECall`. We do not construct the dynamic invocation sequences, as doing so requires heavy static analysis costs [23]. `Uranus-dep` is also used in `loadClass(C)` (§4.1) to compute dependent classes of class `C` at runtime.

6 EVALUATION

6.1 Case Study

We integrated Uranus with Spark [76] and Zookeeper [13] to build two a privacy-preserving big-data computation platform (Spark-Uranus) and a privacy-preserving co-ordination service (ZooKeeper-Uranus). Details of our modifications to the two softwares are shown in Appendix D.

For Spark-Uranus, we run only the user-define-functions (UDFs) within enclaves to preserve the confidentiality and integrity of user data and computed results. We added annotations to three functions of the Spark framework. Each annotated function decrypts input data and encrypts computed output. To preserve the integrity of UDFs, the annotated functions use Uranus's `loadClass` API to load and verify UDF's dependent classes. To preserve the execution integrity of Spark DAG (Direct Acyclic Graph), a sequence of UDF, we adopted the self-cation protocol in Opaque [77].

ZooKeeper-Uranus preserves confidentiality and integrity of data in ZooKeeper. ZooKeeper-Uranus adopts trusted code partition presented in SecureKeeper [28], an SGX system that customizes ZooKeeper to preserve data confidentiality and integrity. ZooKeeper-Uranus adds annotations to three functions, which decrypt data received from client requests and encrypt data after parsing it. ZooKeeper's file path names are encrypted using a deterministic encryption approach, and payloads are encrypted with path names to prevent attackers replacing a path's data.

Opaque's workload [Dataset]	Opaque's size	Uranus's size
Filtering [Rankings]	1.1 Million	0.9 Billion
AdvRevenue [UserVisits]	1.2 Million	1.8 Billion
RevenueAggr [UserVisits]	1.2 Million	1.8 Billion
PageRank [Friendster]	10 Million	1.8 Billion
LinearRegr [Linear regression]	1.0 Million	1.8 Billion
PatientsQuery [Diseases]	0.5 Million	0.2 Billion
TreatmentQuery [Treatments]	0.5 Million	0.2 Billion
GeneQuery [Genes]	0.5 Million	0.2 Billion

Table 1: Queries and dataset. All Opaque's queries are evaluated by Uranus. All dataset sizes are the number of records.

Overall, Uranus is easy-to-use, as it requires less than 500 LoC, much fewer than the rewriting approach which usually requires adding more than $5k$ LoC to its Java and unsafe C/C++ code (Appendix D). The developments of Spark-Uranus and ZooKeeper-Uranus take two weeks and one week for one researcher, respectively. Most of our efforts in developing the two applications mainly fall in classifying the boundary between the enclave and outside. For example, since executing O_U's member functions are not allowed within an enclave, we have to reconstruct O_U using data passed into the enclave to execute these functions when executing them is necessary.

6.2 Setup and Workload

Our evaluation was conducted on ten computers with SGX-equipped Intel(R) Xeon(R) CPU E3-1280 v6 with 4 cores, 64GB RAM and 2TB SSD. All computers form a cluster with 40Gbps network. For the setup of clients in ZooKeeper-Uranus, we ran the clients in host machines outside the cluster, as clients are trusted and run outside the cloud in our threat model. `ping` latency between clients and servers is 800us. Uranus's enclave heap size is $80MB$ to avoid SGX paging. This heap size setting is common in practice [77].

We compared Spark-Uranus and ZooKeeper-Uranus with two security systems (i.e., Opaque [77] and SecureKeeper [28]) and two native and insecure applications (i.e., Spark [76] and ZooKeeper [13]). We ran Spark-Uranus with all applications using Uranus's JIT compiler (§4.2) by default. We used ten machines for each query, and each machine runs one enclave with four threads by default. For Spark-Uranus, we included all 8 queries evaluated in Opaque.

We did not compare Spark-Uranus with VC3 [59] because it is close-source. VC3 obtains its performance overhead in an SGX simulator, so we did not compare our results with VC3's. We compared Spark-Uranus and Opaque's encryption mode [77], which provides the same security guarantee as Spark-Uranus. Therefore, the comparison between Spark-Uranus and Opaque's encryption mode is apple-to-apple. Opaque uses the code-rewrite approach, among existing SGX-based big-data systems [11, 54, 59, 77], Opaque's encryption mode [77] is the most efficient system in terms of dataset sizes and performance overhead. Opaque also has an oblivious mode to handle CPU architectural access pattern attacks, out of the scope of this paper (§3.1). For SGX-Spark [11], we compiled its code and found that it is undergoing development. We were unable to compile SGX-Spark in our cluster due to missing code

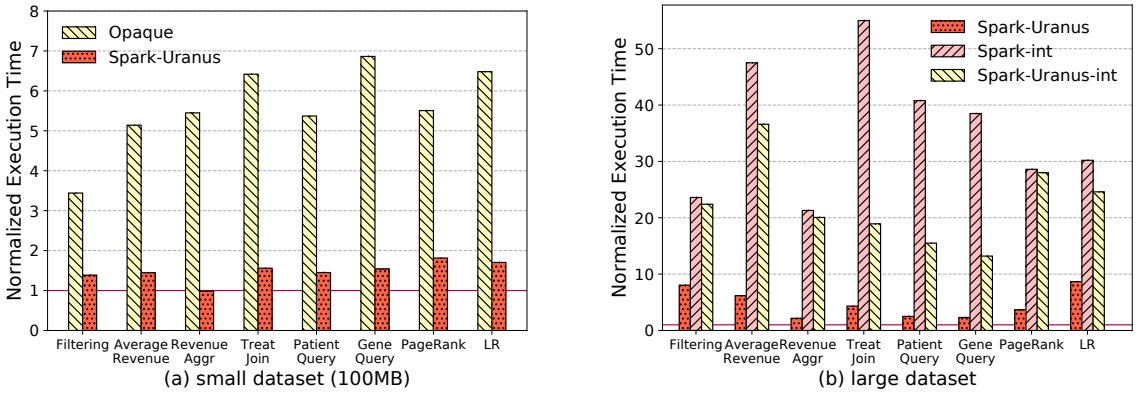

Figure 6: Spark-Uranus performance overhead compared with native Spark and Opaque. "1" (red line) means no overhead.

Program	Total	JECALL /JOCALL		Encryption /Decryption
	Time (s)	Count	Time (s)	Time (s)
Filter	4.28	582	0.01164	0.307
AdvRevenue	6.02	97	0.00194	0.776
RevenueAggr	51.1	5.5k	0.10954	4.382
TreatmentJoin	7.57	185	0.0037	0.074
PatientsQuery	5.32	167	0.00334	0.033
GeneQuery	9.87	425	0.0085	0.085
PageRank	59.8	3.89k	0.07774	2.021
LinearRegre	4.13	97	0.00194	0.093

Table 2: Breakdown of Spark-Uranus (Opaque's dataset).

Program	Total	ECALL /OCALL		Encryption /Decryption
	Time (s)	Count	Time (s)	Time (s)
Filter	23.4	1.2M	3.88	5.37
AdvRevenue	58.6	4.16M	12.40	22.92
RevenueAggr	85.0	5.46M	16.38	29.20
TreatmentJoin	73.8	1.27M	3.83	6.87
PatientsQuery	39.2	0.70M	2.10	3.49
GeneQuery	176.9	1.39M	4.18	7.00
PageRank	185.0	14.1M	42.36	75.59
LinearRegr	33.7	1.2M	3.60	6.15

Table 3: Breakdown of Opaque (Opaque's dataset).

components (e.g., `shm`). Therefore, we did a qualitative analysis between Spark-Uranus and SGX-Spark in §6.3.

For Spark-Uranus, the evaluated 7 dataset is taken from Opaque, including 3 medical dataset and 4 big-data dataset. For all 8 queries, Spark-Uranus was evaluated with typical sizes of dataset as in Spark [76], two to three orders of magnitude larger than the dataset sizes evaluated in Opaque (Table 1). For ZooKeeper-Uranus, we used a popular benchmark ZK-Smoketest [14]. We used concurrent connections to make the servers reach peak throughput. All data points were the median of 11 executions. We focused on these questions:

- §6.3 What is the performance of Spark-Uranus compared to Opaque and Spark?
- §6.4 What is the performance of ZooKeeper-Uranus compared with SecureKeeper and ZooKeeper?
- §6.5 How does Uranus defend against attacks?
- §6.6 What are Uranus's limitations?

6.3 Spark-Uranus v.s. Opaque

Figure 6a shows the performance overhead of Spark-Uranus and Opaque on Opaque's dataset. Spark-Uranus's performance is normalized to native Spark, and Opaque's performance is normalized to native SparkSQL because Opaque is built on SparkSQL. Opaque's implementation can run a maximum 10 million records for all the eight queries (Table 1). We looked into Opaque's code and found that its code restricted dataset size using assertions, and we were unable to make Opaque work with larger dataset even the assertions were removed. With the dataset size Opaque can support, Spark-Uranus is on-average 3.7X faster than Opaque.

To analyze why Spark-Uranus is faster than Opaque, we collected Uranus's and Opaque's runtime micro events in Table 2 and Table 3. By comparing the number of (J)ECalls, Opaque's ECall frequency is much higher. The reason is that Opaque does ECalls for each SparkSQL operator [77]; Spark-Uranus's `JECall` wraps UDF (e.g., `map`), and the call frequency of these functions is proportional to the total number of executed Spark tasks [76]. The number of these tasks is much small than the number of SparkSQL operators in practice. The encryption/decryption time of these queries was negligible except for the first three queries in Table 2, because these queries did fewer computations than the other queries. For example, `Filter` checks only if each record meets a condition provided in UDF. Spark queries are all functional, objects did not escape across threads in an enclave, so Uranus's region-based GC took the fast path (§4.4) and did not incur observable performance overhead. We will evaluate our region-based GC in Figure 7.

Overall, Uranus enables a simple trusted and untrusted code partition for Spark: each UDF is called by a wrapper function annotated with `JECall` (§6.1). Opaque integrates SGX in the SparkSQL layer (i.e., each SQL operator does one ECall), because this can avoid rewriting the readily mature Java UDF (Spark queries) or SparkSQL queries using C/C++. Opaque's design choice makes its ECall frequency much higher than Spark-Uranus's (Table 2).

Because SGX-Spark's code cannot compile in our cluster, we did a qualitative study between Spark-Uranus and SGX-Spark on both performance and security guarantees. To the best of our knowledge, Carlos et al. [19] is the only paper that reports SGX-Spark's performance. This paper runs SGX-Spark Streaming on up to 32MB medical dataset and reports a performance overhead of

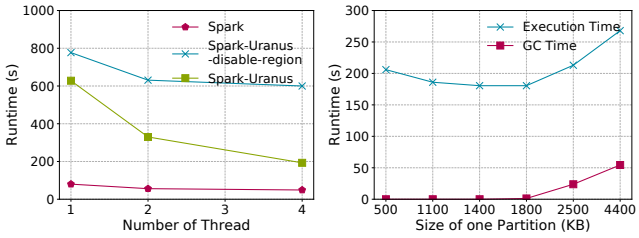

Figure 7: Spark-Uranus's sensitivity on the number of threads per-enclave and input partition size per-thread (LinearRegr).

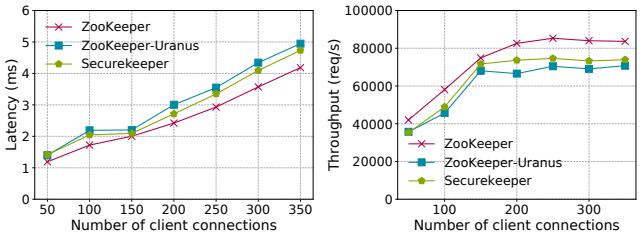

Figure 8: Performance of coordination services.

4X ~ 5X over vanilla Spark. On the similar dataset size (around 100 MB), Figure 6a shows that Spark-Uranus incurs an overhead of 1.7% ~ 80.2%, so Spark-Uranus is much faster than SGX-Spark. Because SGX-Spark runs an unmodified JVM in an enclave, it lacks a protocol to verify the integrity of loaded UDF. Uranus's new bytecode attestation protocol (§4.1) can be integrated in SGX-Spark to achieve the verification task. Uranus's efficient GC (§4.4) can also be integrated in SGX-Spark.

To compare Uranus's interpreter and JIT compiler performance (§4.2), we wrote a simple PI calculation program in Java and ran it in an enclave with the `perf` command in Linux. Uranus's interpreter took 9.9s to finish, while its compiler took only 1.7s. We found that the interpreter incurred 59M missed predicted branch instructions in 3 billion branch instructions, while the compiler incurred 1.6M missed predicted branches in 152M branches. Compared to the interpreter, Uranus's JIT is more efficient, as it greatly reduces missed branches.

Figure 6b shows Spark-Uranus's performance overhead on large dataset. Spark-Uranus incurred 1.2X to 7.6X overhead compared to native Spark on typical large dataset. Spark-Uranus incurred the smallest overhead in RevenueAggr, as it is shuffle-intensive and shuffle code runs outside enclaves.

Comparing Figure 6a and 6b, Spark-Uranus incurred higher overhead when dataset was larger. A possible reason is that native Spark's Hotspot JIT compiler generated more optimized code when execution time was longer, while Uranus's JIT (§4.2) contains no IR optimizations to maintain a small TCB. To validate this reason, we ran Spark-Uranus completely with interpreters both within and outside enclaves (Spark-Uranus-int), and native Spark completely in interpreter (Spark-int), shown in Figure 6b. Spark-Uranus-int has similar performance to Spark-int. This implies that Uranus's JIT is the main reason of Spark-Uranus's performance overhead due to the removal of IR optimizations. Spark-Uranus-int is faster than Spark-int on some queries due to Uranus's efficient region-based GC (§4.4).

Figure 7a investigates the effectiveness of Uranus's GC on multi-threading. We ran Spark-Uranus and Spark-Uranus with Uranus's region-based mechanism disabled (all threads share a global enclave heap). In native Spark, when each machine ran only one thread to process data, Spark's execution time was 80.0s. When each machine ran four threads to process data concurrently, the execution time was 49.1s, a 38% improvement. For Spark-Uranus, when the number of thread increased from one to four, Spark-Uranus's execution time decreased from 628.2s to 193.0s, a 69.2% improvement. When we disabled Spark-Uranus's region-based memory management in each enclave, and when the number of threads in each enclave increased from one to four, Spark-Uranus's execution time decreased from 777.1s to 600.2s, a merely 22.7% improvement. This 22.7% improvement is worse than native Spark's, because enclave memory is merely about 100MB, and frequent GCs were invoked on the enclave heap. Surprisingly, Uranus's region-based GC is more scalable than native Spark's, indicating that Uranus's region-based memory management is suitable for data-handling applications. Actually, Yak [55] has also confirmed a similar scalability benefit when a GC adopts a per-thread memory management for Spark.

Figure 7b evaluates the sensitivity of input data size for each thread (task). This figure suggests that a 2MB partition size for input data is suitable for running big-data queries with SGX. While increasing this size, Uranus consumes more enclave memory, and Uranus's GC starts to invoke MarkSweepCompact when the enclave page pool has no available page (§4.4). The time taken in Uranus's GC grew when the size of each partition increased. Note that enclave memory consumption includes both the partition and its intermediate results. If EPC capacity increases in the future [56], Spark-Uranus's performance overhead would be further reduced.

To analyze the performance overhead of Uranus's runtime checks for enforcing enclave boundary (§4.3) and GC STW (§4.4), we disabled both their checks in all Spark-Uranus queries using the typical dataset sizes (Spark-Uranus queries do not share objects among threads, so disabling GC STW will not affect the executions); Uranus's performance improved by merely at most 1.2% among all queries. For enforcing the enclave boundary, only the trusted bytecode accessing heap objects (e.g., `putfield`) requires injecting runtime checks on the constant enclave memory bounds (§4.3), and the trusted bytecode accessing stack variables (e.g.,`aload`) does not require these checks. Uranus's GC STW injects only checks to function entry points and back-edge of basic blocks, and these checks observe only `needGC` without doing an atomic operation (§4.4). `needGC` remains zero most of the time for real-world applications.

To stress test the overhead of these runtime checks, we wrote a simple Java program, which does only `getfield` and `putfield` in a busy loop within enclaves. This query incurred a 8.9% performance degradation when Uranus's runtime checks were re-enabled. In real-world applications, the percentage of heap-access and back-edge instructions among all trusted bytecode instructions is much lower than the percentage in this simple query, so Uranus's runtime checks incur little performance overhead in real-world applications.

6.4 ZooKeeper-Uranus v.s. SecureKeeper

We ran Zk-Smoketest for ZooKeeper, ZooKeeper-Uranus and SecureKeeper with a typical workload of 30% SET and 70% GET, and make all these systems reach peak throughput. This workload is

evaluated in SecureKeeper's paper. Figure 8 shows the latency and throughput of three systems. ZooKeeper-Uranus had only 19% performance overhead compared with ZooKeeper, which is reasonable considering ZooKeeper-Uranus's security guarantees. SecureKeeper's performance is close to ZooKeeper-Uranus's, because SecureKeeper's code is highly customized and uses little memory in enclaves. SecureKeeper adds around 3.4k LoC C/C++ code to ZooKeeper's Java code base, making ZooKeeper hard to maintain. In contrast, ZooKeeper-Uranus adds only a few `JECall` annotations to ZooKeeper and has the same security guarantee compared to SecureKeeper. Our evaluation (Appendix F) shows that Uranus's GC is efficient even when slow paths exists in an application.

6.5 Attack Analysis.

OS root privileged attacks. An attacker may try to see sensitive data or to tamper with application logic. Uranus tackles these attacks: for data confidentiality, all objects created in Uranus enclave are stored in enclave memory; for integrity, the Uranus runtime (§4) does not take unverified data or code from outside enclaves. Uranus ensures the execution integrity of each `JECall` and its invocation sequence.

Code and data rollbacks. An attacker may try to replace application code with a stall, buggy version to compromise data privacy. Uranus ensures the integrity of each code version, because clients can verify the hash of all dependent classes and manifest (§4.1). An attacker may try to rollback encrypted data outside enclaves. Prior work [53] shows that this attack can be tackled by SGX monotonic counters in enclaves. Uranus tackles this attack by providing API for trusted code to access these counters and to compare their value with the value of the counters in the decrypted data.

Iago attacks. In this paper, we consider Iago attacks that break the type-safety or control flow integrity of the trusted code when running a JVM within an enclave. Specifically, an attacker may manipulate values or object references passed to `JECall` or return results from `JOCall` during enclave transitions to conduct three kinds of attacks. First, an attacker may try to forge a field of O_U in order to change the control flow of enclaves. Uranus prevents such attacks as it forbids enclave code from reading objects outside enclaves, and asks developers to use Uranus's API to read and verify the legitimacy of the content. Second, an attacker may change the pointers stored in a field of O_U to memory stack within enclaves, so that subsequent writes to the forged pointer will poison the call stack. Uranus prevents such attacks, since Uranus's API of reading from O_U checks if the pointer read into enclave is within the enclave memory. Third, an attacker may forge the class of O_U to change the control flow when executing its member functions. Uranus prevents such attacks by forbidding executing O_U's member functions within enclaves. Overall, attackers cannot poison enclave memory during enclave transitions or during proactive/mistaken cross-boundary memory access, so Uranus prevents Iago attacks from compromising type-safety and integrity of the trusted Java code. Verifying the content of `JOCall` to prevent application-level Iago attacks is application developers' responsibility, as Uranus do not know the semantic and actual return results of `JOCall`.

Information leakage via AEX and OCall. An attacker may try to observe the number of AEX or OCall in JVM components during the execution of trusted code. Specifically, he can observe the OS signals thrown from enclaves and infer the number of exceptions or GC during a `JECall`, so he may infer the plaintext data being processed in enclaves. Uranus prevents this leakage by isolating its GC and exception handler components from OS, so that the execution flows of the same trusted code on processing different encrypted data will produce roughly the same numbers of AEX/OCall. Handling applications-level side-channels (e.g., the number of `JOCall`) is developers' responsibility.

Java exceptions and dynamic checks. We tested the robustness of Uranus's dynamic checks (§4.3) by writing buggy code that writes enclave data to outside. Uranus threw an `EnclaveException` with encrypted stack traces without revealing any plaintext. To continue executions of the enclave, we wrote a handler in enclave code for `EnclaveException` that logs the exception and continues handling user requests. The program then continued, wrote encrypted exception logs and returned `null` to us without leaking any sensitive data.

6.6 Limitation and Future Work

Our current implementation of Uranus supports Java and Scala, and Uranus can be extended to run other dynamic languages such as Python [7] and JavaScript [8] in the future, because JVM can interpret these languages. More TEE implementations such as TrustZone enclave [27, 32] and Sanctum [30] can also be integrated in Uranus with proper engineering to gain cross-platform compatibility.

Since Uranus implements an easy-to-verify JIT compiler based on Hotspot's interpreter. Spark-Uranus's performance overhead on large big-data dataset is not negligible (§6.3), because Uranus's JIT does not contain IR optimizations in order to maintain a small TCB. Nevertheless, Spark-Uranus is the first SGX work that has shown to work with typical big-data dataset sizes, and ZooKeeper-Uranus's performance overheads is low. In future work, Uranus's JIT can incorporate easy-to-verify IR optimizations [52] to improve efficiency, and type-safety formal verification [74] for correctness.

Our current evaluation focuses on Spark and Zookeeper, and Uranus can be also used to protect other data-handling applications (e.g., Storm [1]) and web servers (e.g., Tomcat [2]) in the future with proper engineering work. For Spark and Zookeeper, the partition of trusted and untrusted code is already clear in their relevant SGX systems (e.g., SGX-Spark and SecureKeeper). Inferring a partition between the trusted code and untrusted code automatically for an arbitrary application is not the main task of this paper, and static analysis (e.g., Glamdring [51] and [57]) can work as Uranus's orthogonal tools to achieve this task. Uranus's boundary checking protocol can completely enforce the trusted and untrusted code partition at runtime, which can complement unsound assumptions in Java static data flow analysis (§4.3). Differential privacy [35, 49] and data shuffling [77] can also be integrated into Spark-Uranus to provide better data protections.

7 RELATED WORK

TEE. TEE provides a strong confidentiality and integrity guarantees for applications with efficiency, as it effectively removes BIOS, hypervisor, and OS out of TCB. There are diverse TEE implementations such as Intel SGX [38], AMD SEV [42], ARM TrustZone [17], and

IBM SecureBlue [25]. Komodo [32] and Sanctum [30, 64] propose verifiable TEEs on ARM and RISC-V respectively. Timber-V [73] and and Ginseng [75] are two recent TEE implementations for memory efficiency and low-TCB. OpenSGX [39] is an SGX emulator for research. TLR [58] proposes a .NET framework running on ARM TrustZone. TLR is not designed for the client-server model, as it needs a trusted third party to do attestations. Moreover, the secure OS in TLR must be trusted while Uranus assumes an untrusted OS, so Uranus proposes OS-decoupled components to tackle IaGo attacks and side-channel leakage.

SGX-based Systems. SGX systems can be classified into two categories according to the code running in SGX. First, the shielding category (e.g., Haven [21], SCONE [18], Graphene-SGX [67], Pesos [47], and SGX-Kernel [66]) runs all application code in SGX enclave. Second, the customizing category (e.g., Opaque [77], VC3 [59], SecureKeeper [28], S-NFV [61], SGX-Tor [45], Panoply [62], SGX-BigMatrix [60], SGX-Log [43], MiniBox [50], ShieldStore [46], EnclaveDB [56], SGX-Spark [11], and Ryoan [37]) runs only the code processing secret plaintext data in SGX. Uranus belongs to the customizing category, as it annotates and protects sensitive functions. Panoply [62] proposes the abstraction Micron, which provides POSIX API such as multi-threading and multi-processing to run parts of C/C++ applications in enclaves. SGX-BigMatrix [60] is a Python-based secure and oblivious data analytic system for matrix computation. SGXElide [20] loads encrypted code at runtime to provide confidentiality of enclave code, but it does not tackle dynamic loaded code as in Uranus. JIT-Guard [33] uses SGX to protect a JIT compiler, not applications. Several recent systems (e.g., RustSGX [71], GoTEE [34] and ScriptShield [70]) try to run Rust, Go and script languages in enclaves. They focus on specific security challenges in their own languages, including Rust memory safety, secure GO channels and script code compatibility, respectively. Uranus tackles specific security challenges in Java, including dynamic code loading and GC. CordaSGX [4] is an industry project developed for Java applications; currently it evaluates only key management in enclaves. CordaSGX lacks an integrity attestation protocol for dynamic loaded code, and Uranus can help.

Civet [16] is a recent system that automatically partitions and executes Java code. Uranus and Civet are largely complementary in three major aspects. First, Civet uses static analysis for inferring the trusted code partition and requires manual annotation for dynamically loaded code (e.g., reflection). Uranus can help Civet reduce its manual annotation efforts for dynamically loaded code, as Uranus supports loading dynamically loaded code automatically using `loadClass` (§4.1). Second, to enforce a safe enclave boundary, Civet uses dynamic flow tracking (Phosphor [23]) to prevent sensitive data leaking out of the enclave, which has high memory consumption for data-intensive applications and produces prohibited performance and memory overhead when enabling the checking of implicit control flows [40]. Uranus presents Read-integrity and WriteConfidentiality (§4.1), an efficient approach to enforce a safe enclave boundary. Third, Civet's GC protocol optimizes the performance of the traditional generational GC by proposing a three-generation GC to reduce L3 cache misses and EPC page swapping in doing GC. Uranus proposes a region-based GC optimized for data-handling applications, enabling them to run efficiently with typical big-data datasets. Moreover, Uranus tackles information leakage from AEX and OCall, while Civet does not discuss these leakage cases.

Regional GC. Regional-based JVM GC has been used for big-data platforms (Yak [55]), or real-time applications(RTSJ [22, 26]). Uranus's GC differs from these works in two aspects. First, none of these GC systems is fully-automated. For instance, Yak requires developers to provide region hints in Java code. Uranus is fully-automated by leveraging the enclave boundary. Second, and more importantly, these GC protocols are all OS-assisted and not suitable for managing enclave memory.

8 CONCLUSION

We have presented Uranus, an easy-to-use, efficient, and secure SGX programming system for Java applications. Uranus explores a new high-level SGX programming method, which hides the details of low-level TEE implementations. We identified security and efficiency challenges during designing Uranus; we presented two new secure and efficient protocols for dynamic code loading and GC. Extensive evaluations show that Uranus is practical for data-handling applications. Uranus source code and evaluation results are released on https://github.com/hku-systems/uranus.

ACKNOWLEDGMENTS

We thank our shepherd, Lucas Davi, and all anonymous AsiaCCS reviewers for their helpful comments. This work is funded in part by the research grants from Huawei Innovation Research Program Flagship 2018, HK RGC GRF (17202318, 17207117), HK RGC ECS (27200916), and a Croucher innovation award.

REFERENCES

[1] Apache Storm. http://storm.apache.org/.
[2] Apache Tomcat. http://tomcat.apache.org/.
[3] ChronicleMap. https://github.com/OpenHFT/Chronicle-Map.
[4] Corda SGX JVM. https://github.com/corda/sgxjvm-public.
[5] Java ASM Package. https://asm.ow2.io/.
[6] JVM Runtime. http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html.
[7] The Jython Project. www.jython.org.
[8] Multi-threaded JavaScript on the JVM. ringojs.org.
[9] SecureWorker. https://www.npmjs.com/package/secureworker.
[10] SGX-LKL. https://github.com/lsds/sgx-lkl.
[11] SGX-Spark. https://github.com/lsds/sgx-spark.
[12] SpiderMonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.
[13] ZooKeeper. https://zookeeper.apache.org/.
[14] Zookeeper Smoketest. https://github.com/phunt/zk-smoketest.
[15] Handling segfault in SGX enclaves. https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/734597.
[16] Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity20/presentation/tsai
[17] ARM. Security technology building a secure system using TrustZone technology (white paper).
[18] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX.. In *OSDI*. 689–703.
[19] Pierre-Louis Aublin, Peter Pietzuch, and Valerio Schiavoni. Using Trusted Execution Environments for Secure Stream Processing of Medical Data. In *Distributed Applications and Interoperable Systems: 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings*. Springer, 91.
[20] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. SGXElide: enabling enclave code secrecy via self-modification. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 75–86.

[21] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the Eleventh Symposium on Operating Systems Design and Implementation (OSDI '14)*. 267–283.

[22] William S Beebee and Martin Rinard. An implementation of scoped memory for Real-Time Java. In *International Workshop on Embedded Software*. Springer, 289–305.

[23] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 83–101.

[24] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *27th USENIX Security Symposium USENIX Security 18)*. 1213–1227.

[25] Rick Boivie and Peter Williams. SecureBlue++: CPU support for secure execution. *Technical report* (2012).

[26] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee Jr, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 324–337.

[27] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves.. In *NDSS*.

[28] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter R Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX.. In *Middleware*. 14.

[29] Stephen Checkoway and Hovav Shacham. *Iago attacks: Why the system call api is a bad untrusted rpc interface*. Vol. 41. ACM.

[30] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation.. In *USENIX Security Symposium*. 857–874.

[31] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory* 22, 6 (1976), 644–654.

[32] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 287–305.

[33] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. JITGuard: Hardening Just-in-time Compilers with SGX. (2017).

[34] Adrien Ghosn, James R Larus, and Edouard Bugnion. Secured routines: language-based construction of trusted execution environments. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 571–586.

[35] Xueyang Hu, Mingxuan Yuan, Jianguo Yao, Yu Deng, Lei Chen, Qiang Yang, Haibing Guan, and Jia Zeng. Differential Privacy in Telco Big Data Platform. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1692–1703.

[36] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'10)*.

[37] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data.. In *OSDI*. 533–549.

[38] Intel. Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/329298-001.pdf.

[39] Prerit Jain, Soham Jayesh Desai, Ming-Wei Shih, Taesoo Kim, Seong Min Kim, Jae-Hyuk Lee, Changho Choi, Youjung Shin, Brent Byunghoon Kang, and Dongsu Han. OpenSGX: An Open Platform for SGX Research.. In *NDSS*.

[40] Jiang Jianyu, Zhao Shixiong, Alsayed Danish, Wang Yuexuan, Cui Heming, Liang Feng, and Gu Zhaoquan. Kakute: A Precise, Unified Information Flow Analysis System for Big-data Security. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC '17)*.

[41] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel® Software Guard Extensions: EPID Provisioning and Attestation Services. *White Paper* 1 (2016), 1–10.

[42] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper* (2016).

[43] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. SGX-Log: Securing system logs with SGX. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 19–30.

[44] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. Closing the gap–the formally verified optimizing compiler CompCert. In *SSS'17: Safety-critical Systems Symposium 2017*. CreateSpace, 163–180.

[45] Seong Min Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments.. In *NSDI*. 145–161.

[46] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 14.

[47] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*.

[48] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*. 523–539.

[49] Tsz On Li, Jianyu Jiang, Ji Qi, Chi Chiu So, Jiacheng Ma, Tianxiang Shen, Heming Cui, and Amy Wang. UPA: An Automated, Accurate and Efficient Differentially Private Big-data Mining System. In *To appear at the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '20)*. IEEE.

[50] Yanlin Li, Jonathan M McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. MiniBox: A Two-Way Sandbox for x86 Native Code.. In *USENIX Annual Technical Conference*. 409–420.

[51] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17), Santa Clara, CA*.

[52] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. *ACM SIGPLAN Notices* 50, 6 (2015), 22–32.

[53] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. *IACR Cryptology ePrint Archive* 2017 (2017), 48.

[54] Saeid Mofrad, Ishtiaq Ahmed, Shiyong Lu, Ping Yang, Heming Cui, and Fengwei Zhang. SecDATAVIEW: a secure big data workflow management system for heterogeneous computing environments. In *Proceedings of The 35th Annual Computer Security Applications Conference (ACSAC'19)*.

[55] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 349–365.

[56] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A Secure Database using SGX. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. IEEE, 0.

[57] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *ICSE*. IEEE, 923–934.

[58] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 67–80.

[59] 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 *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 38–54.

[60] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS '10)*.

[61] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing NFV states by using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 45–48.

[62] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *Proc. of the Annual Network and Distributed System Security Symp.(NDSS)*.

[63] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe* 2007 (2007).

[64] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2435–2450.

[65] Yogesh Swami. Intel SGX Remote Attestation is not sufficient. *IACR* (2017).

[66] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. SGXKernel: A Library Operating System Optimized for Intel SGX. In *Proceedings of the Computing Frontiers Conference*. ACM, 35–44.

[67] 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)*.

[68] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1741–1758.

[69] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM, 4.

[70] Huibo Wang, Erick Bauman, Vishal Karande, Zhiqiang Lin, Yueqiang Cheng, and Yinqian Zhang. Running Language Interpreters Inside SGX: A Lightweight, Legacy-Compatible Script Code Hardening Approach. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. ACM, 114–121.
[71] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2333–2350.
[72] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security*. Springer, 440–457.
[73] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V.
[74] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. *ACM Sigplan Notices* 45, 6 (2010), 99–110.
[75] Min Hong Yun and Lin Zhong. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System.. In *NDSS*.
[76] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX Association, 2–2.
[77] 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 *NSDI*. 283–298.

9 APPENDIX

A JAVA BYTECODE SIZE

Package Name	Dependent Classes Size	Jar Size
LinearRegr (UDF)	1.1 KB	1.1 KB
Spark + Scala	2.3 MB	17.4 MB
Java RT	5.0 MB	65.0 MB
Total	7.3 MB	82.4 MB

Table 4: Enclave memory usage of Java bytecode.

B SAMPLE JIT CODE

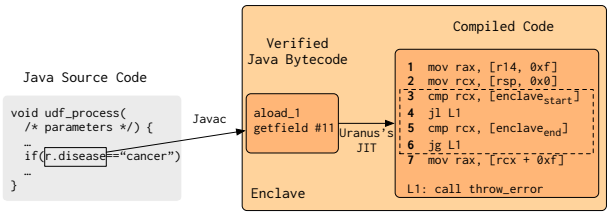

Figure 9: Uranus's JIT Compilation.

Figure 9 shows that verified Java bytecode sequences are compiled into native code. `aload_0` and `getfield` are compiled as instruction 1 and instruction 2 ~ 7, respectively. Figure 9 shows the native code with runtime checks (in dash box, line 3 ~ 6).

C URANUS'S MULTI-THREADING AND EXCEPTION HANDLING

Uranus supports multi-threading executions and in-enclave synchronization. Uranus does not support thread creation within an enclave, so Uranus is a one-to-one mapping between an enclave thread and an external thread. Uranus re-implements Java's locking mechanism with spinlocks, as these locks do not require going across enclave boundary. We does not choose to block threads in locks, as blocking requires help from OS and leaving the enclave, which makes synchronization bugs easier to exploit [65, 69, 72]. An `EnclaveException` is thrown when synchronization (e.g., `lock` and `wait`) is invoked on non-enclave objects. Uranus's current multi-threading support is sufficient for our evaluation.

To handle exceptions, Uranus adopts a similar design of Hotspot's exception capture and dispatch mechanism. Hotspot makes use of OS signals and dynamic checks to capture exceptions. For example, Hotspot does not explicitly check if an object is `null` for `NullPointerException`. Instead, it proceeds such memory accesses, which incurs `segfault` for a Hotspot-defined handler to capture the faults. Uranus did not adopt this exception capturing mechanism using OS signals for two reasons. First, SGX cannot safely handle memory faults in enclaves. Specifically, although current SGX hardware can handle some hardware faults (e.g., `sigfpe`) by using AEX, the SGX hardware does not provide sufficient information (e.g., `segfault` memory address) for memory faults within an enclave [15]. Therefore, handling memory faults can relay on only the exception information provided by OS, insecure for Uranus: an attacker outside an enclave can manipulate the exception information to tamper with the control flow of the trusted Java code running within the enclave. Second, it may result in information leakage, since handling OS signals requires leaving enclaves, and attackers can infer control flows or plaintext data by observing if there are exceptions.

Uranus supports handling memory exceptions in enclaves, including `NullPointerException`, `ArrayOutOf BoundException` and `ArithmeticException`, by using runtime checks. Some of these checks (e.g., array bound checks) are already in JVM, and Uranus adds only `NullPointer` checks and `DividedByZero` checks. When an exception is captured by Uranus, Uranus searches for a handler of the exception in the current function, and searches in the caller if such one does not exit, recursively. When Uranus cannot find a corresponding handler for an exception in enclaves, Uranus throws an `EnclaveException` outside enclaves with exception information (e.g., exception location) encrypted. This set of exception support is sufficient to run the trusted code of the real-world applications in our evaluation.

D EVALUATED APPLICATION

Framework	LoC Modifications
Distributed Data Analytics:	
Opaque (encryption mode)	4k (C++), 2.6k (Scala)
VC3	7k (C/C++)
Spark-Uranus	4 annotated functions and encryption/decryption [250 LoC]
Privacy-preserving ZooKeeper	
SecureKeeper	3.4k (C), 154 (Java)
ZooKeeper-Uranus	2 annotated functions and encryption/decryption [87 LoC]

Table 5: Code modified by Uranus and code rewriting systems.

Table 5 shows the number of LoC added to Spark-Uranus and ZooKeeper-Uranus. For Spark-Uranus, we add three wrapper functions, annotate them with `JECall`, and use these functions to call Spark UDF in `ResultTask.run()`, `ShuffleMapTask.run()`

and `SortShuffleWritter` of Spark. The first two wrappers call Uranus's `loadClass` API to load and verify UDF's dependent classes. Specifically, `ResultTask` and `ShuffleMapTask` execute `map` and `reduce` functions, respectively; `SortShuffleWritter` merges reduced data before shuffling data. To preserve the execution integrity of Spark DAG (Direct Acyclic Graph), a sequence of UDF, we adopt the self-verification protocol in Opaque [77]. The driver program of Spark-Uranus computes an authentication message $\texttt{Auth} \leftarrow \texttt{Encrypt}_k(\texttt{id}, \texttt{DAG}, \texttt{P}_1, ..., \texttt{P}_p)$, where `id` is the stage id, $\texttt{P}_i$ is the partition id. A task execution function verifies Spark DAG's integrity by checking the authenticity of `Auth`. On the other hand, the `JECall`-annotated functions encrypt processed results along with the authentication message `Auth`. Spark-Uranus clients verify `Auth` within encrypted results.

ZooKeeper-Uranus annotates two functions which process user data: `FinalRequestProcessor.processRequest()` and `PrepRequestProcessor.pProcess()`. These functions decrypt data received from client requests and encrypt data after parsing it. ZooKeeper's file path names are encrypted using a deterministic encryption approach, and payloads are encrypted with path names to prevent attackers replacing a path's data.

E COMPARISON OF LOC

System	Component	LoC
Uranus	JIT compiler	14,411
	Garbage Collector	6,600
	Code Verifier	1,281
	Exception Handler	310
	Native Libraries*	6,837
	Bytecode and Class*	22,494
	SGX SDK	171,606
JVM on SGX-LKL [10]	JVM*	913,951
	SGX-LKL	38,870
	SGX-MUSL	99,222
Civet [16]	Civet's components	38,481
	Modified JVM	422,199
	Graphene-SGX	49,689
	GNU libc*	1,008,773

Table 6: Comparisons of Loc between Uranus, SGX-LKL-JVM and Civet. Components with ∗ are ported from the codebase of GNU libc or JVM and are not modified.

Table 6 shows the comparisons of LoC between Uranus, JVM running on SGX-LKL and Civet. For Civet, we use the LoC reported in Civet's paper [16]. For JVM on SGX-LKL, we measured its LoC in its repository [10]. Overall, Uranus has much fewer LoC than the other two systems. Moreover, most of Uranus's LoC is from SGX SDK, and Uranus's codebase can be further reduced by removing some SGX SDK's components that are not used by Uranus.

F MORE BENCHMARKS OF URANUS'S GC

Except for the stress test of Uranus's GC in Figure 7b, both Spark-Uranus and ZooKeeper-Uranus take the fast path of Uranus's GC because their threads running enclaves conduct computation individually and do not share objects. To analyze Uranus's GC

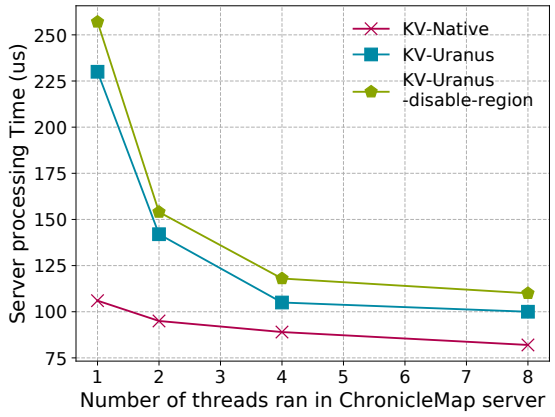

Figure 10: KV-Uranus's multi-threading scalability.

efficiency in slow path (§4.4), we used the key-value store program in Figure 1 and the ChronicleMap [3] library to implement KV-Uranus, a secure key-value store server that parses the plaintext of key-value pairs only in enclaves. To reduce the time cost of encryption and decryption for every request, we implemented a plaintext key-value pair cache shared by all threads within the enclave. Each key and value is 16 byte respectively, and the cache is 10MB, including key-value pairs and metadata. Outside an enclave, the value and key are encrypted together to avoid attackers replacing a key's value with another's. KV-Uranus takes the same partition as SecureKeeper and ZooKeeper-Uranus, and thus has the same security guarantees.

In Figure 10, we ran KV-Uranus with the same ZK-Smoketest [14] benchmark and same workload as for ZooKeeper-Uranus. We measured the median processing time on the KV-Uranus server program (i.e., network round-trip time was excluded in order to precisely analyze Uranus GC's effect) in three settings: KV-Uranus; KV-Native, the native and insecure key-value server program; and KV-Uranus with Uranus's region-based GC mechanism disabled (all threads share a global enclave heap). We varied the number of threads in the key-value server program. When the same number of threads are run on the server, KV-Uranus's per-request processing time was 10us ∼ 22us, faster than KV-Uranus-disable-region's. We found that the total GC time for all threads of KV-Uranus and KV-Uranus-disable-region were 12.5s and 21.4s, respectively. Therefore, we believe Uranus's GC is efficient for diverse applications.