Civet: An Efficient Java Partitioning Framework for Hardware Enclaves

Chia-Che Tsai
Texas A&M University

Jeongseok Son
UC Berkeley

Bhushan Jain
UNC Chapel Hill

John McAvey
Hendrix College

Raluca Ada Popa
UC Berkeley

Donald E. Porter
UNC Chapel Hill

Abstract

Hardware enclaves are designed to execute small pieces of sensitive code or to operate on sensitive data, in isolation from larger, less trusted systems. Partitioning a large, legacy application requires significant effort. Partitioning an application written in a managed language, such as Java, is more challenging because of mutable language characteristics, extensive code reachability in class libraries, and the inevitability of using a heavyweight runtime.

Civet is a framework for partitioning Java applications into enclaves. Civet reduces the number of lines of code in the enclave and uses language-level defenses, including deep type checks and dynamic taint-tracking, to harden the enclave interface. Civet also contributes a partitioned Java runtime design, including a garbage collection design optimized for the peculiarities of enclaves. Civet is efficient for data-intensive workloads; partitioning a Hadoop mapper reduces the enclave overhead from $10\times$ to 16–22% without taint-tracking or 70–80% with taint-tracking.

1 Introduction

Hardware enclaves [1–4] are designed to protect sensitive code and data from compromised OSes, hypervisors, or off-chip devices. An enclave includes a memory region protected by the CPU and encrypted in DRAM. An enclave can also attest the integrity of execution to a remote entity. So far, many enclave-protected systems have been proposed [5–8], including commercial cloud offerings from Microsoft, IBM, and Alibaba [9, 10]. Speaking broadly, there is an increasing understanding of how to use enclaves to protect a single client's code in a multi-tenant cloud.

The design space for enclaves quickly becomes murkier for complex cloud applications that contain sensitive and insensitive components, and that are written in an object-oriented, managed language. These applications often integrate large code bases and data from both users and cloud providers, who may distrust each other. Take Hadoop [11] as example:

	Ubuntu 16.04	Graphene-SGX	
	Total time (s)	Total time (s)	(Δ)
Mappers	45.4 +/- 0.5	501.0 +/- 9.4	(**10.0$\times$**)
Reducers	35.2 +/- 0.2	393.1 +/- 8.1	(**10.2$\times$**)
Garbage collection	1.9 +/- 0.0	14.8 +/- 0.9	(**6.6$\times$**)

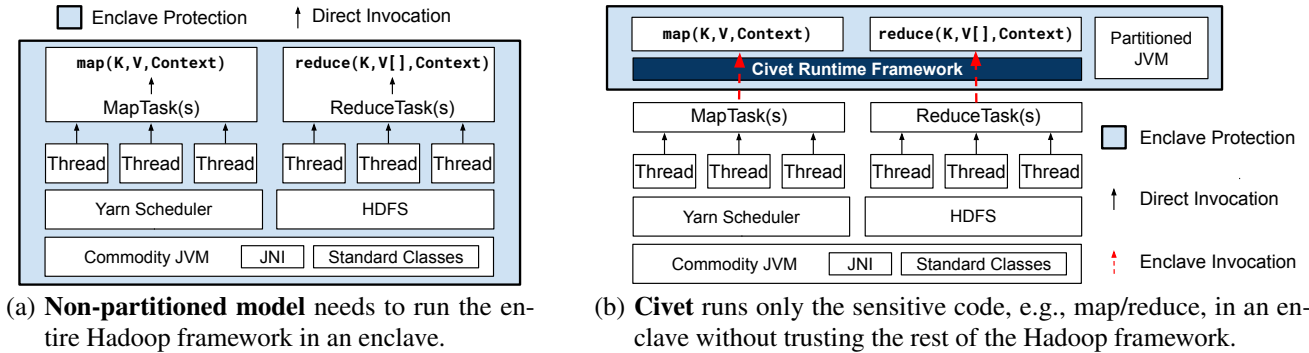Table 1: Comparison of a non-partitioned Hadoop job between Ubuntu and Graphene-SGX [16]

user-defined mappers and reducers may operate on sensitive data, yet the orchestration and resource management framework is controlled by the cloud provider. Although there are some solutions for running an entire application in an enclave [12–15], this approach provides no isolation between the user and cloud provider. Moreover, dropping an entire cloud framework written in a managed language like Java into an enclave is prohibitively expensive, as illustrated in Table 1. Experiment parameters are detailed in §9.

Ideally, an application like Hadoop should be **partitioned**, so that only sensitive code and data are inside the enclave. Figure 1(a) illustrates the non-partitioned model for protecting the entire Hadoop framework. The model places a large portion of framework code in the trusted computing base (TCB), despite the fact that this code need not directly interact with any sensitive user data.

This paper presents **Civet**, a framework for partitioning a Java application into trusted classes that run inside enclaves, and untrusted classes that run outside enclaves. Figure 1(b) shows Civet's partitioned model, which reduces the in-enclave TCB to sensitive classes. The partitioned model establishes a hardware-enforced isolation boundary between the untrusted "system" and trusted pieces of application logic within a large, legacy code base.

Prior work [17–19] has explored the idea of partitioning an application for enclaves, yet no solution can partition a Java application that depends on complex class libraries and a complex runtime. Among prior work, TLR (Trusted Language Runtime) [17] is a framework for running portions of a mo-

Figure 1: A comparison between the non-partitioned model and Civet's partitioned model.

(a) **Non-partitioned model** needs to run the entire Hadoop framework in an enclave.

(b) **Civet** runs only the sensitive code, e.g., map/reduce, in an enclave without trusting the rest of the Hadoop framework.

bile application, written in C# , inside ARM's TrustZone [20]. Although TLR provides a mechanism for separating sensitive logic from the untrusted OS and application code, mobile applications are much simpler than most cloud applications. TLR provides no solutions for hardening the trusted code against Iago-style attacks [21] that leverage subtle language properties such as polymorphism. Intel's Software Guard Extensions (SGX) [1], a more common platform for emerging cloud deployments of hardware enclaves, has a much tighter memory budget than TrustZone; this memory restriction can be especially problematic for Java workloads. Glamdring [18] is another framework for automatically partitioning C/C++ programs into enclaves. Glamdring reduces the TCB using program slicing, but does not generate code to protect against malicious inputs. In our experience, a key challenge in partitioning a legacy application is hardening the software at the newly created enclave boundary.

Civet addresses various challenges of partitioning a managed, object-oriented language, using Java as a representative example. Our framework is prototyped on SGX, but many of the design principles are independent of SGX.

1.1 Challenges

To partition a Java application, developers face several challenges that *reduce* security compared to the original application, that fail to reduce the TCB, or that require memory and other resources in excess of the constraints of SGX. We identify the following challenges for partitioning an application written in a managed, object-oriented language, such as Java:

- *Complexity of defending partition interfaces:* Adding an interface between trusted and untrusted code requires adding a defense; this is already a challenge, but the language features of Java further complicate this defense. With polymorphism, untrusted code may override the behavior of a method by creating a subclass. By accepting objects from outside the enclave as input, an enclave can become potentially vulnerable to a **type confusion** attack [22]. The input can be subtyped to alter the behavior of the enclave

code, with an overridden method potentially sending sensitive data out of the enclave, or using reflection to load unexpected code into the enclave.

- *Large application code footprint:* Even a "Hello World" class can introduce millions of lines of code from standard and third-party libraries. Many classes rely on JNIs (Java Native Interfaces), which are written in C/C++ and are notoriously prone to vulnerabilities [23]. Finally, a feature-complete JVM like OpenJDK contains up to a million lines of code written in Java and C/C++.

- *A runtime that requires significant resources and system support:* Even a small partition of a Java application needs a full-featured runtime. Designing runtimes for enclaves is an open problem—a commodity JVM like OpenJDK makes many assumptions that are violated by enclaves, such as the presence of a large, demand-allocated virtual memory and a large pool of internal maintenance threads. Standard runtime behaviors, such as garbage collection, are not tuned for the memory restriction of SGX.

1.2 Goals and Contributions

To address these challenges specific to supporting managed languages in enclaves, Civet includes both compile-time tools and an execution framework with the following goals:

- *Reducing partitioning effort:* When introducing an isolation boundary into a large codebase, reasoning about the resulting security implications can be challenging— including what code ultimately runs in the enclave, what data can enter and exit the enclave, and by what code paths. To assist the developer in this reasoning process, we add static analysis and dynamic code instrumentation tools that can both reduce the code footprint in the enclave, as well as give the developer visibility into what can run in the enclave, data ingress, and data egress.

- *Mitigating partitioning pitfalls:* Partitioning can expose a larger attack surface than running the entire application inside enclaves. A goal of Civet is to mitigate a majority of

the *non-side-channel* security pitfalls caused by partitioning, such as type confusion attacks or accidental leakage through data flow. To this end, Civet analyzes the application and applies restrictions to behaviors that are impossible before partitioning. For type confusion attacks, we present an efficient strategy for type-checking any input, not only at the root of an object, but at every field and array element. Civet also uses taint-tracking [24] to block outputs that are tainted by sensitive information.

- *Removing unreachable code:* Even in a managed language, unreachable code in the TCB is a potential liability, as dynamic class loading or polymorphic behavior can lead to invisible or unexpected execution paths. During offline analysis, Civet removes unreachable classes and methods. The result is a trusted JAR file that is significantly smaller than the original collection of classes libraries, improving the auditability and lowering the risk of unexpected behaviors in the enclave.

- *Optimizing garbage collection for enclaves:* SGX has a hardware limitation of 93.5 MB for the Enclave Memory Cache (EPC). If the enclaves on a system access more DRAM than this, the OS will swap the memory in and out of EPC, causing substantial overhead [13,25]. Most GCs scan the heap and, thus, perform poorly when the heap is sparsely populated and is larger than the EPC. Civet includes a GC design that adds a middle generation, for preventing full-heap GC while keeping GC faster for the youngest objects. This optimizes GC to match the performance characteristics of enclaves.

The contributions of this paper are:
- A framework that leverages Java language features to analyze and partition applications to run in enclaves (§4).
- A system to harden the enclave boundary. This includes type-checking polymorphic inputs (§5), and mitigating unintended information leakage from enclaves (§6).
- A lightweight JVM partitioned for enclaves (§8).
- A study of GC and a three-generation GC design optimized for enclaves (§7.2).

2 Related Work

Enclave frameworks and SDKs. Intel SGX introduces new design challenges, such as validating system call results from a malicious OS [21]. The state-of-the-art solution is a library OS [12, 16] or a shield layer [13, 26] to hoist OS functionality into the enclave and/or validate inputs from an untrusted OS. Developers can also write enclave code from scratch, using an SGX SDK [27–29]. Applications written in a managed language are commonly rewritten for SGX in another language; for example, VC3 [5] sacrifices the benefits of using a type-safe language and compatibility by rewriting the Hadoop code in C++.

Partitioned trusted execution. Prior work reduces trusted code size through program slicing and/or generating the interface between partitions. TLR [17] and Rubinov et al. [30] partition android programs to run in ARM TrustZone [20]. Glamdring [18] partitions C/C++ programs for enclaves using static program slicing. SeCage [19] partitions an application into secret compartments with hardware-based isolation. GoTEE [31] compiles Go functions into enclaves, with a lightweight runtime and APIs for shielding. Brenner et al. [32] run microservices in enclaves, apart from the orchitestration framework. EnclaveDom [33] leverages Memory Protection Keys (MPK) for privilege separation inside enclaves.

Java partitioning frameworks. A number of tools partition a Java application for modularity. Addistant [34] and J-Orchestra [35] automatically divide Java applications across multiple hosts or JVMs. Zdancewic et al. [36] use annotations to partition an application, with static analysis to enforce data flow policies. Swift [37] partitions web applications such that security-critical data remains on the trusted server.

Capability languages such as E [38], Joe-E [39], Oz-E [40], and Emily [41] define the object-capability approach for various languages, and identify patterns for secure programming. Compared to these capability-based frameworks, Civet enforces coarse-grained security policies by simply separating trusted and untrusted objects, and hardening the boundary with hardware enclaves.

3 Threat Model and Security Properties

Civet adopts a similar threat model to many recent SGX projects [5, 12–15, 18, 26, 31]. All in-enclave software is trusted and everything else that is outside the enclave is not trusted. Because any software can have bugs, which an attacker could exploit, one of Civet's goals is to decrease the TCB running in the enclave, as well as reduce the attack surface of the enclave code exposed to the untrusted host.

In moving from a model where one can trust the OS and hypervisor, to an SGX-style threat model, where host software and even parts of the application are potentially compromised, one must design enclave code to resist several new threats. First, one must ensure that the code in the enclave is really what the authors intended. Although SGX can measure the contents of an enclave *at start time*, the enclave code itself must be responsible to handle dynamic loading of additional classes; one cause for concern is misleading the enclave code to load a malicious class that could leak sensitive data or compromise the integrity of the code in the enclave. Second, partitioning an application to run portions of code in an enclave creates a new intra-application interface. Although good software engineering involves explicating assumptions about the state of the application when a function is called, perhaps even as comments, one must now carefully check these assumptions at the enclave boundary. This general class

of semantic attacks against an enclave interface that violate a tacit assumption in the code are called Iago attacks [21]. A third major concern is that sensitive data not inadvertently leak from the enclave. In refactoring a large piece of legacy code, it is easy to accidentally leave a code path that writes data to an out-of-enclave object. This third concern is less of an attack vector per se, so much as an aspect of this work that is highly error-prone. The security properties discussed later in this section consider each of these concerns.

Untrusted components. An attacker can compromise any off-chip devices (e.g., DRAM, accelerators, I/O devices) and any code running outside the enclave, including the host OS, system software, or hypervisor.

Trusted components. Civet trusts the CPU and any other hardware in the CPU package, as well as any binaries running inside the enclave. The enclave will include the trusted Java classes, the in-enclave JVM, the remaining trusted JNIs, Graphene-SGX, GNU libc, and Civet's in-enclave framework.

We assume attackers have the source code of the application and Civet, and may attempt Iago-type attacks [21] by manipulating inputs to enclave interfaces, including class-level, JNI-level, and system-level APIs. For system-level APIs, Civet inherits shielding code from Graphene; inasmuch as a Civet partition *extends* the enclave attack surface with class-level interfaces, Civet adds additional, language-based defenses on the data ingress and egress of the enclave.

Out-of-scope attacks. Civet assumes a correctly implemented CPU. Civet does not protect against known limitations of current enclave implementations like Intel SGX, which include rollback attacks [42], micro-architectural vulnerabilities [43, 43, 44, 44–49], cache timing attacks [45, 50, 51], and denial-of-service from the host. Solving these problems is orthogonal to the contributions of Civet.

Balancing TCB and Attack Surface. Compared to running an entire application in an end-to-end shielded framework [12, 13, 15, 16], partitioning an application has the advantage of reducing the TCB that directly interacts with sensitive code and data, as well as minimizing enclave footprint (important for performance on current enclave hardware). However, partitioning introduces new attack surface between the application code inside and outside the enclave. In a framework that shields a POSIX-style interface, one can simply use an existing shield that protects against many issues, such as Iago attacks [21]. When one designs a custom enclave interface after partitioning a large code base, one has to design shielding code between code that was originally mutually trusted.

A key goal of Civet is to help developers harden this new enclave interface. For application-level vulnerabilities, Civet requires the developers to design defenses for the interaction between the trusted classes inside the enclave and the untrusted classes outside the enclave, but provides language tools to help developers reason about these defenses, such as injection of *shield classes* and taint-tracking. Civet hardens the partitioned JVM for developers, and inherits shielding of OS-level interfaces from Graphene-SGX.

We note that partitioning an application can also potentially introduce new side-channel vulnerabilities. Side channels and their defenses are out of the scope of this paper.

Security properties. Civet is designed to enforce the following security properties:

- *I–Code integrity and remote attestation*: Civet checks the integrity of all code running inside the enclave, including the Java classes, Java virtual machine, imported Java Native Interface (JNI) libraries, system libraries, and the Graphene-SGX library OS. A remote entity can use the remote attestation feature of hardware enclaves to check the measurement of a Civet application. This property is fundamental to hardware enclaves and is necessary for defending against code modification or code injection attacks.

- *II–Type integrity on enclave interfaces:* Polymorphism at the enclave interface causes confusion for developers when writing shielding code. Civet ensures that the inputs to a method exported as an enclave interface cannot be arbitrarily subtyped as classes that are impossible in the original application. With type integrity on enclave inputs, developers can safely use object-oriented APIs for semantic checks or cryptographic protections. This property is necessary for preventing the type confusion attacks described in §5.1.

- *III–Explicit data declassification*: Data provisioned from a secure channel or derived from this provisioned data inside the enclave cannot be copied outside the enclave unless explicitly declassified by the developer. Civet tracks both the *explicit* flows from operating on tainted objects and, optionally, the *implicit* flows from branching based on tainted values. Developers need to either encrypt or sanitize a tainted object for declassification, or the object cannot return to untrusted code. This property is to prevent semantic bugs in application or defense code from accidentally leaking the secrets from the enclaves. Side channels and other implicit flows are out of scope.

4 Partitioning Class Libraries

In this section, we explain how to partition an application with Civet; and how Civet creates a concise, robust partition with little input from the developer.

4.1 The Partitioning Workflow

Step 1: Identifying enclave interfaces. To create a partition, Civet requires developers to identify one or more **entry classes** within the application, to serve as the interface between enclave code and non-enclave code. Figure 2 illustrates partitioning a Hadoop mapper with an entry class.
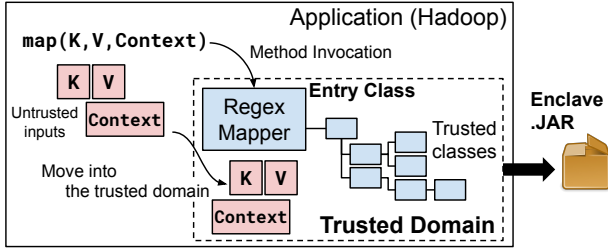
Figure 2: Partitioning model of Civet. The entry classes define a trusted domain inside an application, with all the trusted classes collect into a JAR file.

We note that many other partitioning systems involve specifying sensitive *data* rather than defining a *code* interface; we selected the code option in part because one use case for SGX is protecting sensitive algorithms, and in part because we believe that this approach better matches programmers' intuition. We leave a more careful study of this design choice for future work.

A set of entry classes define a **trusted domain**, in which all of the classes that implement the enclave functionality are mutually trusted. Every call from an untrusted class to an entry class transitions execution into the enclave.

At build time, Civet packs all of the *trusted* classes into a single JAR file, named as *enclave.jar*, which contains all of the Java code that can be loaded into the enclave. The input to this tool is a configuration written in XML (illustrated in Figure 3), with each entry class listed as an <EntryClass> rule. The resulting JAR file can be audited and signed by developers. For a class loaded by reflection, Civet relies on the developer explicitly white-list the class, by adding an <Include> rule to the configuration. Users can add <Include> rules gradually when encountering resolution errors during testing, or search for dynamically-loaded classes in the enclave code. The use of reflection is extremely common in commercial Java applications [52]. Parallel to this work, many papers [52–54] have shown that the usage of reflection calls can be estimated by static analysis. For identifying trusted classes (§4.2), Civet also requires the user to specify the main class of the whole, untrusted program, using an <MainClass> rule.

Step 2: Specifying enclave protections. After defining the entry classes, the developers can specify extra **shield classes** that leverage object orientation to wrap the entry classes. Shield classes are primarily used for tasks such as sanitizing or decrypting inputs, or encrypting outputs. Developers can write a shield class without changing the source code of the original application. Figure 3 shows an example of a shield class for the Hadoop mapper partitioned in Figure 2. RegexMapperShield is a wrapper to RegexMapper for decrypting the inputs and encrypting the outputs. A shield class is defined in the configuration using a <ShieldClass> rule,

```
<EntryClass>RegexMapper</EntryClass>
<MainClass>Grep</MainClass>
<ShieldClass>RegexMapperShield</ShieldClass>
<Include>com.sun.crypto.provider.AESCipher</Include>
<Include>com.sun.crypto.provider.PCBC</Include>
<Declassify>AESCipher.encrypt</Declassify>
```

```
class RegexMapperShield<K> extends RegexMapper<K> {
  Cipher cipher; // Initialized in the constructor
  public void map(K k, Text v, Context context) {
    cipher.init(Cipher.DECRYPT_MODE,
        Enclave.getSealKey(), new IvParameterSpec(k));
    v = new Text(cipher.doFinal(v.getBytes()));
    super.map(k, v, context); // Call the actual mapper
    // Further encrypt the context if necessary
  }
}
```

Figure 3: The configuration (in XML) and shielding class for partitioning a hadoop mapper (RegexMapper).
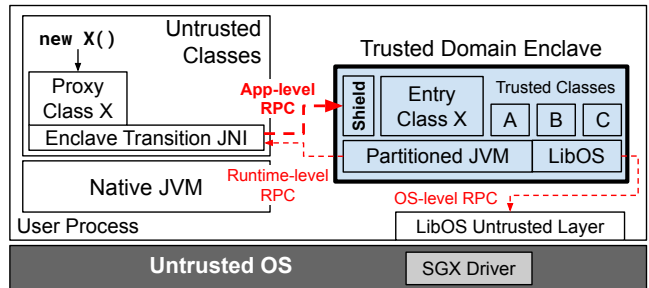

Figure 4: Components of the Civet framework. Civet maps entry classes to a trusted domain inside enclaves. The untrusted code accesses the trusted objects through RPCs by invoking proxy classes. Each enclave also contains a partitioned JVM (§8.2) and a library OS (LibOS).

and as a benefit, the definition is transparent to the entry class as well as the call sites in the non-enclave code.

Civet also synthesizes extra protections, including type-checking inputs and dynamic taint-tracking. Civet defines a **sensitive object** to be an object instantiated inside the trusted code or provisioned from a secure channel. A <Declassify> rule can specify a method to declassify the outputs to the untrusted domain. If an output is not declassified, Civet uses dynamic taint-tracking (§6.2) to block any object from leaving the enclave if the object contains information derived from a sensitive object.

Step 3: Connecting trusted and untrusted domains. For each entry class, Civet synthesizes a **proxy class** that marshals inputs to the enclave and invokes an RPC to code running in the enclave. Figure 4 shows the components of the Civet framework, including a proxy class. An untrusted application, such as the Hadoop framework, can create an enclave by instantiating the proxy classes. A proxy class includes JNI to invoke the hardware level operations to enter an enclave. The code of the entry class runs inside the enclave.

The underlying JVM or library OS may exit the enclave only to 1) return from an application-level call into an entry class, and 2) to implement runtime-level or OS-level functionality. Developers need only concern themselves with the first case. For the second case, the JVM and the library OS include their own shielding code.

Specifically, Civet disallows enclave Java code to call out to non-enclave Java code, which we call a **nested exit**, for two reasons: (1) Designing shielding strategies for nested exits can be challenging; (2) A nested exit exports intermediate states outside the enclave and increases the risk of data leakage, corruption, and side channels. The downside is that the enclave may include more supporting trusted classes and/or export more entry classes for the untrusted code to access results inside the enclave. All of our application examples (§9) were easily partitioned without nested exits.

4.2 Identifying Trusted Code

A key service Civet provides for developers is creating a single JAR file with all of the code that should be reachable from the entry classes or that is white-listed with an include directive, but no other code.

In the presence of polymorphism, this analysis is best done with an automated static analysis. For example, Hadoop uses an interface called `Writable` to represent 52 different types of data. Polymorphism multiplies the complexity of the security analysis, and obscures the implications of bringing a class into an enclave. In a source-code-level audit, developers cannot easily predict the target of every method call or field access. Our analysis helps by generating an unambiguous collection of classes and methods as the transitive closure of control and data flows from entry classes.

Civet determines the classes and methods to be included in the trusted domain via static bytecode analysis:

- **Call graph analysis** [55,56]: For each method, identifying the classes and methods referenced.
- **Points-to analysis** [57–59]: For each field or local variable, identifying the heap objects that are assigned, to determine all the possible subtypes allocated for the field or local variable if it is polymorphic.

We implement the static analysis described in Algorithm 1 using SOOT [60], the de facto bytecode analysis framework for Java. We use the flow-insensitive, context-insensitive, whole program analysis implemented in Spark [61], the points-to analysis framework of Soot, with on-the-fly call graph analysis (see the configuration in §9.4). The points-to analysis is based on the main class specified by the user. We use the points-to analysis to identify the possible argument types to an entry method, or the possible targets of a polymorphic method. For classes that are not included in the whole program analysis of Spark, such as classes explicitly loaded by the JVM during initialization, we conservatively estimate the points-to targets by considering *all* subclasses.

Algorithm 1: Static analysis for identifying trusted code

/* **Extending the entry classes with input types** */
Data: A set of entry classes E and included classes I
1 **while** E *is different from the last iteration* **do**
2   **for** $c \in E$ **do**
3   **for** $m \in$ **public methods of** c **do**
4   **for** $o \in$ **non-primitive arguments of** m **do**
5          $E \leftarrow E \cup \text{classes(points-to}(o))$

/* **Collecting required classes for the enclave** */
6 $Classes \leftarrow E \cup I; CG \leftarrow \emptyset;$
7 **while** $Classes$ *is different from the last iteration* **do**
8   **for** $c \in Classes$ **do**
9   **for** $m \in$ *methods of* c **do**
10   **for** $o.f \in$ **field accesses in** m **do**
11   **if** o *is a class* **then** $OC \leftarrow \{o\}$
12   **else** $OC \leftarrow \text{classes(points-to}(o))$
13          $FC \leftarrow \text{classes(points-to}(f))$
14          $Classes \leftarrow Classes \cup OC \cup FC$
15   **for** $o.m' \in$ **method calls in** m **do**
16   **if** o *is a class* **then** $OC \leftarrow \{o\}$
17   **else** $OC \leftarrow \text{classes(points-to}(o))$
18          $Classes \leftarrow Classes \cup OC$
19          $CG \leftarrow CG \cup \{(c,m,c',m')|c' \in OC\}$

/* **Shredding unreachable methods** */
20 $Methods \leftarrow \{(c,m)|c \in E \cup I, m \in$ **public methods of** $c\}$
21 **while** $Methods$ *is different from the last iteration* **do**
22   **for** $(c,m) \in Methods$ **do**
23   **for** $(c,m,c',m') \in CG$ **do**
24        $Methods \leftarrow Methods \cup (c',m')$

25 **return** $(Classes, Methods)$

Shredding unreachable methods. We incorporate a new technique called **Shredding** to eliminate code that is unreachable at compile time. Shredding is different from partitioning or program slicing because it does not change the control flow of the enclave, and is more similar to dead code analysis [62].

We shred both classes and methods within the class to reduce the footprint of enclave code. By shredding methods, we can subsequently remove classes and methods which are only used inside the unreachable methods. As described in Algorithm 1, the analysis starts with entry classes and classes listed by the <Included> rules, and then recursively includes methods that are reachable inside the enclave. With points-to analysis, we can conservatively identify methods that are possible callees of a polymorphic invocation to a generic class or an interface.

Static fields. The one exception to strict enclave isolation is that enclave code in Civet may access static fields and methods outside of an enclave. If a trusted class access a static field or calls a static method inside the enclave, Civet

includes the target class inside the enclave. If a static field is directly updated by another trusted class, Civet allows this update to propagate out of the enclave, assuming it does not violate any taint-tracking rules.

4.3 Security Discussion

Civet measures the integrity of the code included in enclaves (*Property I–Code Integrity and Remote Attestation*). For each partition/trusted domain, Civet generates a trusted JAR containing signed classes and binaries. Each entry (a file or a directory) in the JAR is securely hashed, with the list of entries and hashes signed by the developer's private key. This prevents subsequent modifications of the JAR by anyone else. The signature of each class is checked by the in-enclave Java runtime, whereas the signature of each binary is checked by the Graphene-SGX library OS. The trusted Java runtime will only load classes and binaries from the trusted JAR.

5 Shielding Polymorphic Interfaces

This section explains how exposing a polymorphic, object-oriented interface can lead to a type-confusion attack, and an efficient type-checking scheme for reducing the risk.

5.1 Type Confusion Attack

Partitioning an application exposes a new attack surface at the interface between the trusted and untrusted code. In the case of OS-level interfaces, such as system calls, this led to an initially surprising and, subsequently, widely explored topic of Iago attacks [13, 14, 21, 26]. In a partitioned Java application, where objects are passed into the enclave as inputs, the complex behavior of polymorphic object-orientation is ripe for Iago-style attacks. Specifically, attackers may pass a polymorphic object as part of the input to the enclave code. This can take the form of creating an object that violates class invariants, or generating control flow that is not possible in the original application.

Attack example: Tomcat. Figure 5 shows an example of how a partitioning choice in an application can leave the enclave open to attack, in this case in a partitioned Tomcat servlet [63]. This example is hypothetical, and selected for clarity, as real-world examples may be more complex and obscure. A Tomcat servlet typically receives a `Request` object that stores the parameters of an HTTP request. For convenience, Tomcat stores the POST message body in a `CoyoteInputStream` object, i.e., a buffered stream, for the servlet to read. A developer might decide to use a generic class at the enclave interface, say changing the requirement from a `CoyoteInputStream` to a generic `inputStream` class. The code behavior is equivalent, and the interface is arguably more flexible. However, an attacker can replace the `CoyoteInputStream` with a subclass of `InputStream`,

```
class HttpResponder extends HttpServlet {
  public void doPost(HttpServletRequest req,
      HttpServletResponse resp) {
    InputStream inputStream = req.getStream();
    byte[] body = new byte[inputStream.available()];
    inputStream.read(body); // Read POST body
    resp.getWriter().write(body);
}}
class Request extends HttpServletRequest {
  InputStream inputStream // This line is changed
    = new CoyoteInputStream(new InputBuffer());
  public InputStream getStream() { return inputStream; }
}
```
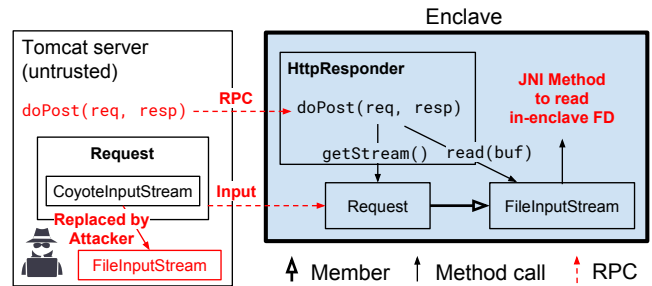

Figure 5: An example in which a servlet (`HttpResponder`) is partitioned into an enclave. An attacker can exploit the polymorphic input of `HttpResonder` to force the class to read from a shielded in-enclave file descriptor.

as long as this subclass is in the trusted domain. For example, this request may be directed to a `FileInputStream` object that is connected to a file that include sensitive data, and could be exfiltrated by serving the request.

In general, this type of vulnerabilities is caused by partitioning the code such that a precondition or invariant is established by code that ends up outside of the enclave. For instance, in a monolithic application, one might have the invariant that one only adds a stream to the `Request` class with one of a few specific subtypes by auditing the instances of `new`, rather than putting redundant assertions at every single method boundary. When selecting a partition interface, it is easy to place these invariant checks in the untrusted code. To the extent that we can statically extract these invariants, Civet can automatically harden the enclave interface.

5.2 Deep Type Checks on Enclave Inputs

In order to harden the enclave interface, Civet automatically generates deep type checks on input objects. Civet uses marshalling, or serialization, to pass input objects into enclaves, and the enclave runs memory bounds checks on these input buffers. In order to prevent possible type-confusion attacks, Civet also implements a deep type check at the enclave boundary. In the case of a complex object with other objects nested underneath it, the enclave checks not just the type of the "root" object, but also the type of every field or array element in the object. A simple cast check (i.e., checking whether an

object is castable to a type) or a type comparison (e.g., "if (o.getClass() != String.class) ...") is insufficient for preventing this type of attacks.

We assume that, if the user is partitioning an application, the untrusted code is initially benign but may be compromised. Thus, to generate type checks at the enclave boundary, Civet currently uses the source code or byte code of the untrusted portion to infer the set of subtypes that could be passed to a given enclave API function could in the original, unpartitioned code. We call this set of types (and field subtypes) for a given object a **profile**. We use this information to generate the type checking code; it would be possible for an expert developer to manually create this information if they did not wish to mine it from application code.

One challenge is that this naïve representation of a profile can grow exponentially large when an object contains a deep hierarchy and many fields at each level. Worse, if a class contains references to itself, or forms cyclic references among multiple classes, the profile can grow indefinitely large. Self and cyclic class references are common in practice.

Path-based type-checks. Instead of defining which types can be part of an input, Civet defines which parts of an input (permission *object*) that a type (permission *subject*) can be instantiated and assigned to. For each type that can be instantiated during input deserialization, Civet lists all the fields and array elements that can be instantiated as the type. These fields are represented as *paths*, as traversed from the root object. The strategy is similar to a mandatory access control (MAC) system, such as AppArmor [64] which has a default deny policy, and the administrator can give a program explicit access to files with certain path patterns. This strategy makes it easy to make permission decisions sooner if the prefix of the path does not match the policy.

We explain the type checks with the example in Figure 5. Assume the static analysis determines that the original application only assigns the `CoyoteInputStream` class to the `inputStream` field of the input, of class `Response`. Civet will generate the rules for instantiating this input:

- For `CoyoteInputStream`:
    - `((Response)req).inputStream`
- For `Response`:
    - `req` (root object)

Based on these rules, any instantiation of a class that does not match its rule will be rejected by Civet. For example, if a `FileInputStream` object is assigned to `inputStream` of `req`, the instantiation will be rejected because the class is not permitted with the given path.

This scheme is efficient for objects with a complex structure. For example, in Hadoop, a `TupleWritable` object contains an array of other `Writable` objects, including another `TupleWritable` object. If we want to reject nested tuples but allow tuples of `LongWritable` and `Text`, the following rules will enforce such a policy:

- For `LongWrtiable` and `Text`:
    - `value` (root object)
    - `((TupleWritable)value).values[*]` (array elements if root object is a tuple)
- For `TupleWritable`:
    - `value` (root object)

Array sizes and indices are indistinguishable in this scheme, hence the wildcard (`[*]`) in the second rule for `LongWrtiable` and `Text`. Extending or re-ordering the elements of an array does not increase the number of rules.

Complexity. We show that the path-based representation simplifies type-checking. Assume that a class contains N fields, and each field can be assigned to one of M subtypes. The number of rules at the first level is O(MN), which is significantly fewer than O(M^N) in the simpler representation. If we consider an object of D levels, the complexity of our scheme is O(MN^D), also much simpler than O(M^{ND}).

Implementation. At build time, we assign a unique identifier to each field of a class that is both: (1) a trusted class, and (2) instantiated and assigned as part of an input to a method. Our prototype uses a 32-bit identifier on the assumption that a partitioned application will not have more than 2^{32} fields among all trusted classes, and could increase this limit if needed. To compare the conditions, we generate a hash of all the fields that have been visited from the root object. Note that the hash must be collision-resistant, otherwise the attacker may submit a malicious structure that collides with a permitted hash. Ideally, we need to use a strong hash function, such as SHA256; however, we observe that most objects in our use cases never go deeper than 8 levels. Therefore, we just push the field identifiers into a 32-byte buffer, and only hash the buffer when the depth is larger than 8.

Compatibility. False negatives in the static analysis may cause compatibility issues if a benign input is rejected by type-checking. Our static analysis only excludes inputs that were impossible in the original application. Among our application examples (§9), no benign input from the original partitioned code was rejected.

5.3 Security Discussion

The deep type checking described in this section ensures *Property II—Type integrity* for enclave interfaces. Specifically, Civet uses static analysis to generate a set of polymorphic types that could happen in the original program, and checks that only objects (or object hierarchies) within that set are accepted as enclave inputs.

A limitation of the type checks is that we need to conservatively approve input types based on the points-to analysis, as well as overestimate classes loaded via reflection or loaded internally by the JVM. This limitation leads to false positives, in which Civet may permit an unexpected input type to an entry method, which may be exploited for type confusion

attacks. We did not observe this issue in our case studies.

6 Declassifying Enclave Outputs

In this section we discuss the security challenges of explicitly declassifying all outputs that can be potentially tainted by sensitive data (*Property III—Explicit data declassification*).

6.1 Data Leakage

Preventing data leakage is a critical challenge for partitioning. When data is decrypted and processed inside an enclave, it is important that the data does not inadvertently make its way back to the untrusted classes, except via explicit declassification. For instance, a privacy-preserving function inside the enclave may report safe results with differential privacy [65]. Developers of partitioned enclave applications have an additional burden of auditing the code for any paths that might leak sensitive data outside of the enclave.

Polymorphism makes it difficult to simply inspect the code statically or an object dynamically, and know whether it was derived from sensitive bits. Developers do not necessarily know whether invoking a method on an ObjectType calls the method of its Class or the Subclass, which in turn may or may not update a field in the object. A further challenge for determining the data flow is the detection of the *implicit* data flow under the effect of the control flow. Since polymorphism and reflection also complicate the control flow, it becomes even harder to predict the data flow of a Java application without a dynamic taint-tracker [24, 66–73]. Therefore, we argue that it is important to track both explicit and implicit data flow within the enclaves that operate on sensitive data.

6.2 Dynamic Taint-Tracking

To ensure data confidentiality, Civet tracks data flows using Phosphor [74], a dynamic taint-tracking framework. In Civet, all the entry class objects and methods of shield classes are marked as taint *sources*. Thus, all the objects which are derived from instantiation of the entry classes or from shielding, such as decrypted data or data provisioned from a secure channel, will be tainted. Phosphor propagates the taints through explicit data flow, and optionally through implicit data flow based on control flow. We added Phosphor as a phase of the partition tool to instrument the classes in `enclave.jar` (§4.1) after shredding. We run the Phosphor instrumenter with the `multiTaint` option, and the `controlTrack` option if the users choose to track the implicit flow.

Dynamic taint-tracking prevents developers from introducing vulnerabilities via buggy code that inadvertently leaks sensitive data through data flows. The *sink* of the taint-tracking is the function for marshaling returned objects, in order to block any tainted object from being flowed out of the enclave, At the boundary of the enclave, any tainted object unless the ob-

ject is explicitly declassified. We modify Phosphor such that developers can specify a `Declassify` rule that can remove taints on objects that are confirmed to contain no sensitive data. In practice, we expect the developers to declassify an object after sanitizing the object or encrypting the data.

We note that tracking implicit data flow is considerably more expensive than tracking explicit flow; thus, we give the user an option to disable this in a deployment run. Because this is a tool primarily for understanding code behavior, there are scenarios where this trade-off is sensible; there are also scenarios where users will prefer more exhaustive checks.

6.3 Security Discussion

Dynamic taint-tracking complements the language safety of Java by requiring any sensitive data that leaves the enclave to be explicitly checked (*Property III–Explicit data declassification*). The JVM ensures that sensitive code and data inside the enclave remain in a hardware-protected memory region. Taint tracking can catch cases where an output derives from sensitive information, but the results were not encrypted or checked against a different policy. We assume the developer writes a declassifier that enforces appropriate application-level policies.

7 Garbage Collection Optimization

Garbage collection (GC) is an essential feature of Java and many managed languages. GC unburdens the programmer from writing error-prone memory management code. GC design and implementation of has a first-order impact on application performance, yet off-the-shelf GC does not perform well in enclaves. Civet contributes an optimized GC design for the constraints of enclaves.

7.1 GC Design Challenges

The Civet JVM prototype is based on the OpenJDK 8 HotSpot JVM, which uses a generational GC [75]. The HotSpot JVM contains multiple GC implementations, each with different advantages and resource requirements. In initial attempts to run Java in an enclave, we found that *no* garbage collection strategy performed well within the constraints of SGX enclaves. Thus, we started with a relatively straightforward GC that we could understand and tune to work within an enclave. Specifically, we studied and tuned the **Serial GC** from HotSpot—a "textbook" generational GC.

In Serial GC, the JVM typically divides the heap into two generations: the *young* (*defNew*) and *old* (*tenured*) generations. The GC strategy is different for each generation, illustrated in Figure 6. The young GC happens frequently to recover memory from short-lived objects. Objects that have survived several GC rounds in the young generation are promoted to the old generation. Specifically, the young gener-
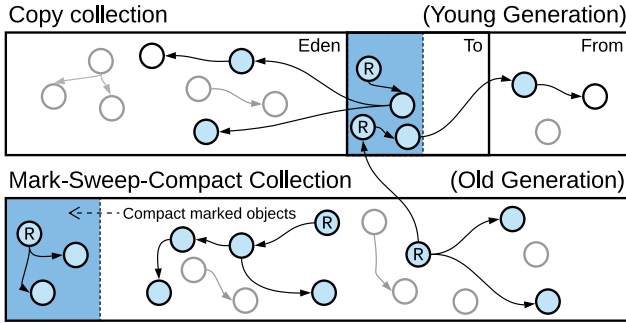
Figure 6: Two garbage collection approaches used in Serial GC. A *Copying* approach evacuates living objects to a reserved space, whereas a *Mark-Sweep-Compact (MSC)* approach separates the phases of discovering live objects from heap compaction.
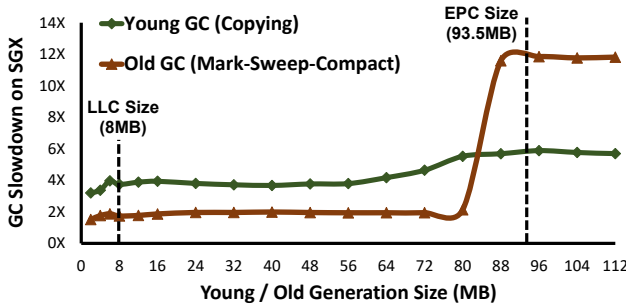

Figure 7: Single-threaded, serial GC slowdown caused by SGX, within the young generation (copying GC) and the old generation (MSC), in respect to different generation sizes. For each GC iteration, 80% of the objects are garbage-collected, while the remaining are compacted or promoted.

ation uses a copying GC that traverses the heap and copies live objects into a reserved space (called the *To* space) on the fly. The underlying assumption is that the live objects will be few, and it is simpler to just copy them than managed fragmented free space. In contrast, the old generation uses a Mark-Sweep-Compact (MSC) strategy, which consists of multiple passes through the heap, and is optimized to minimize movement of objects that are likely to survive.

We observe several problems for both the young and old GCs in enclaves. We illustrate the issues using a simple microbenchmark that targets a 20% object survival rate for both generations, by repeatedly allocating and freeing a forest of 5KB binary trees (each with 31 nodes), occupying 1MB of the heap. Figure 7 shows the average slowdown on each GC iteration in the young and old generations, as a function of different generation sizes. We observe that the Copying GC in the young generation has more slowdown in enclaves until the generation size reaches ∼80MB, due to more LLC misses during data movement. Note that LLC misses in enclaves are expensive, as they involve decrypting and integrity check
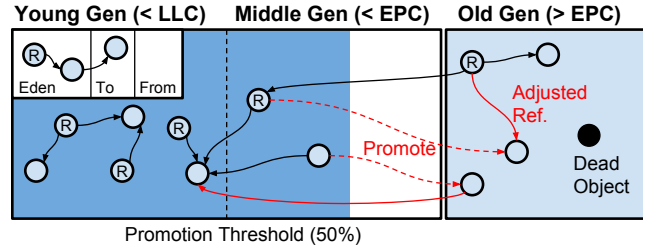

Figure 8: Civet proposes a GC strategy, with a middle generation as a middle ground before promoting object to old generation. The middle GC follows a partial promotion strategy, with an adjustable threshold.

for the data [13, 76]. When the generation size is close to or larger than the EPC size, the slowdown on MSC becomes significantly higher then the Copying GC due to an order-of-magnitude higher number of page faults, which are even more expensive than LLC misses.

We observe three performance regimes for GC, which reflect the underlying hardware limitations:
- If the generation fits inside the LLC (8MB on Intel E3-1280 v5), copying GC is even more efficient than MSC.
- If the generation fits in the enclave page cache (EPC—the protected physical memory that is used inside enclaves), the cost of GC is proportional to the size of the generation.
- When the generation size approaches the EPC size, MSC becomes much worse than copying GC because of EPC swapping. Currently, the EPC is limited to 93.5 MB of usable memory; after this is exhausted, the OS must swap the encrypted contents of the EPC to other DRAM or disk. Some of the EPC must be used for the code and stack, so there is an upward trend closer to 80 MB.

Prior work [13, 25] reports up to a 1000× slowdown for random reads and writes in an enclave larger than the EPC. This size limitation has not been enlarged on any later generations of Intel CPUs. Because MSC-based GC traverses the heap more times than the copying GC, it will incur more swapping when the GC'ed space exceeds the EPC.

7.2 GC Optimization for Enclaves

The experiment above indicates three distinct performance regimes for enclaves. Thus, we adopt a three-generation design, where each generation has a target working set size: (1) smaller than the LLC size (8MB), (2) between the LLC size and the EPC size (93.5MB), and (3) larger than the EPC size. The goal of this three-generation design is to minimize cache misses in the young GC and the page faults in the old GC. For the rest of the paper, we refer to these as the *new*, *middle*, and *old* generations.

Figure 8 illustrates our three-generation GC design. The middle generation adopts the same MSC strategy as the old generation. Objects that survive the young generation get a
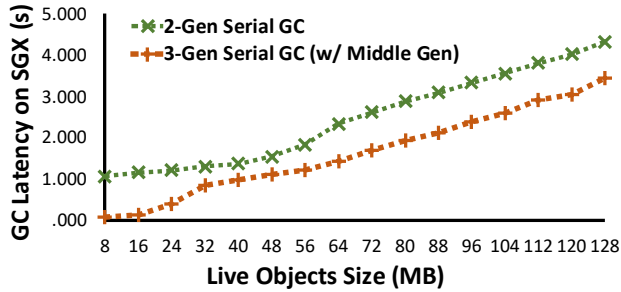
Figure 9: Average GC latency (including all generations) on SGX, in regards to the total live object sizes. The comparison is between two-generation and three-generation serial GCs. The heap size is 256 MB, with the young and middle generations at 2 MB and 48 MB, respectively.

second chance of being reclaimed in the middle generation, before being promoted to the old generation. Similar to the young GC, the middle GC also walks the references from the known roots, but does not traverse the unclaimed dead objects in the old generation. This keeps the middle GC from accessing objects outside of the EPC boundary and reduces the number of page faults incurred by the GC.

To keep short-lived objects in the middle generation longer, we set a promotion threshold to decide which objects should be promoted to the old generation. The middle GC only promotes objects when the size of the remaining live objects surpasses a promotion threshold (e.g., 50% of the generation). The promotion threshold is adjustable by users.

We further reduce memory accesses outside the enclave by leveraging the *remember set* abstraction in the HotSpot VM. We also noticed that after MSC, the *adjust reference* phase scans the entire heap to identify and adjust references to a compacted or promoted object, causing significant cache misses and EPC swapping The *remember set* use a coarse-grain bit map to track the region which contains recently promoted objects to scan for references to younger generations. Our JVM updates the remember set during the marking phase of GC, so that the middle GC only has to scan memory regions that are known to contain references.

We implement our GC strategy on HotSpot JVM and measure the impact of the middle generation on the average GC time. Figure 9 shows the average GC time of two-generation and three-generation GCs. We use a randomized allocation workload and adjust the total size of live objects, which reflects the effective space used by the application. The total heap size is 256 MB and the young generation size is 2 MB. Based on our tuning, the best size of the middle generation is 48 MB and the remaining space is the old generation. The results show that with our GC, the average GC time (including middle and old GCs) is consistently 0.5–1.0 seconds faster than Serial GC (20–89% improvement), at all live object sizes.

8 Runtime Implementation

This section describes the implementation of the Civet runtime framework.

8.1 Civet Runtime Framework

Given the entry classes, our partitioning tool automatically generates the RPC interfaces for entering and leaving enclaves. The generated interfaces primarily serve two purposes: (1) intercepting invocations to entry classes and seamlessly converting them into RPCs, and (2) marshaling and verifying the input and output objects for the entry classes. To reduce the execution-time TCB and to improve RPC latency, Civet directly generates bytecode for the RPC interface and supporting code inside the enclave.

For marshaling objects in and out of enclaves, Civet uses the Fast-serialization library [77], or fst (v2.50), instead of using the built-in serialization API. fst generates a more compact representation of each object; for instance, at runtime, fst allows Civet to register all the classes needed for marshaling both inside and outside the enclave, so that object types can be represented numerically instead of as strings. Furthermore, we use the off-heap serializer of fst, which reduces the instantiation cost of marshaling buffers and reduces GC during RPCs. The off-heap buffers are allocated per in-enclave worker thread, and are reused throughout the enclave execution.

8.2 Reducing Framework TCB

The Civet framework contains several trusted components, shown in Table 2. Civet includes a modified JVM, based on OpenJDK 8 HotSpot runtime, which has a smaller TCB and fits into the memory limitation of enclaves. This is a preliminary effort—there are additional opportunities to further shrink or partition the JVM:

- Garbage collector: Civet removes most of the garbage collectors, such as G1GC and parallel scavenge GC, and only keeps an optimized serial GC (§7.2).
- Compiler: the default option in Civet is ahead-of-time compilation (AOT). AOT is time-consuming (∼20 minutes to compiling 4,000 classes), but introduces no overhead to the execution. For users who cannot compile the bytecode ahead of time, Civet provides the options of including the C1 (platform-generic) and/or C2 (architecture-specific) compilers in the enclave; or using only the interpreter. The former increases the in-enclave TCB, whereas the latter introduces significant overheads (10–1000×).
- JVM-related classes: A large portion of the JVM functionalities are implemented in Java classes. We can simply use static analysis to include the classes needed by the TCB and shred the others. Table 2 does not include these classes.
- JNI libraries: Finally, a large portion of the C++ code in the OpenJDK code base contributes to the JNI library, such as

Civet components (language):		Total LoC	
Partition tool (Java)		3,611	
Runtime framework (Java)		2,166	
Runtime JNI (C++)		1,093	
Phosphor framework (Java)		31,611	
Modified runtime components:	**Original**	**Partitioned**	**(Δ%)**
JVM (`libjvm`)	593,159	303,826	(49%)
JNI (`libjava`, `libzip`, ...)	423,303	68,684	(84%)
Graphene-SGX	55,974	49,689	(11%)
Unmodified runtime components:		**Total LoC**	
GNU libc 2.19		1,008,773	

Table 2: The complexity of the whole Civet framework and the run-time TCB measured in LoC (lines of code), including both modified and unmodified components.

`libjava`. We observe that a portion of the JNI library, especially the system-tier functionality, is perfect for partitioning outside the enclave. For example, `FileInputStream` contains native methods to read a file. These JNIs are originally shielded by Graphene-SGX, but can be moved outside the enclave to reduce the TCB.

In total, Civet removes 49% of the JVM code and 84% of the JNI code from the trusted computing base. To access OS functionality from the enclave, Civet uses Graphene-SGX and GNU libc, which could be further reduced in code size.

9 Case Studies and Evaluation

In this section, we evaluate the efficiency of Civet using three use cases, to show the sensitivity of the TCB and performance to the partition boundary chosen by the developers. We select three applications that accept user-provided code in a somewhat modular design. Each of these applications varies in the degree to which the interface for user-provided code matches what should run in the enclave, and thus, the degree of difficulty in partitioning. In the case of Tomcat (§9.2), users provide code at a granularity very close to what should go in the enclave. In the cases of and Hadoop (§9.3) and GraphChi (§9.3), the users provide code, but issues such as batching inputs to the enclave require a more careful decision about partitioning boundaries.

We also evaluate the cost of static analysis and the breakdowns of performance overheads using microbenchmarks. Unless otherwise noted, we configure Phosphor's taint-tracking to only track explicit flows; tracking implicit flows typically adds $10\times$, which dwarf other overheads from Civet.

All experiments are collected on a Supermicro SYS-5019S-M server. The CPU is a 8-core 3.70 GHz Intel Xeon E3-1280 CPU, with microcode patched for Spectre mitigation. Out of 32GB RAM on the machine, 93.5MB is dedicated to enclaves. The system runs Ubuntu 16.04.4 LTS server with Linux kernel 4.15.0-58-generic, with Page Table Isolation
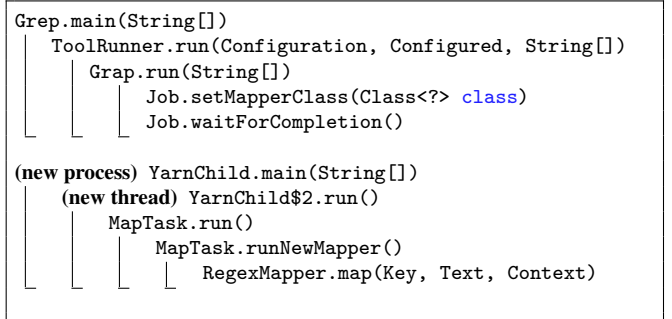
```
Grep.main(String[])
  ToolRunner.run(Configuration, Configured, String[])
    Grap.run(String[])
        Job.setMapperClass(Class<?> class)
        Job.waitForCompletion()

(new process) YarnChild.main(String[])
    (new thread) YarnChild$2.run()
      MapTask.run()
        MapTask.runNewMapper()
          RegexMapper.map(Key, Text, Context)
```

Figure 10: The call graph in Hadoop with `RegexMapper`.

Selected entry methods	Shredding	#C	#M	LoC	Δ%
Before partitioning		68.5K	589.7K	7.2M	
① `MapTask.*`	class	12.9K	115.3K	1.5M	79%
	method	4.3K	20.7K	372.5K	95%
② `RegexMapper.*`	class	4.2K	38.0K	509.2K	93%
	method	2.1K	12.1K	247.8K	96%

Table 3: Partitioning results of Civet for Hadoop, partitioned with two boundaries and measured in classes (#C), methods (#M), and lines of code (LoC). For both cases, `AESCipher` and `PCBC` are explicitly included for dynamic loading.

(PTI) enabled. The Civet implementation is based on Open-JDK v1.8.0_71, Phosphor v0.0.4 [24], Intel SGX Linux SDK and driver v2.3 [78], and Graphene-SGX v0.6 [14].

9.1 Hadoop

Hadoop [11] is a widely used framework for distributed computing and big data. We choose the regular expression parser (`RegexMapper`) as an example, but the usage can be generalized to other Hadoop applications. Running regular expression parsing inside enclaves is beneficial for protecting sensitive data that might be processed in a distributed manner, such as system or network logs.

Hadoop already has a modular architecture, and is easily partitioned with Civet. Coarse-grained partitioning at the main function is not practical, because Hadoop is multi-process, illustrated in Figure 10. A more natural division point is within a worker (or process): ① `MapTask.run()` as a generic boundary that can include any mapper; ② `RegexMapper.map()` as the mapper class itself. Although the former is more generic, the latter can have a smaller TCB.

Figure 11 shows the execution time of searching a regular expression inside a large, encrypted authentication log (1GB), using `RegexMapper` as the partition boundary. The sample is encrypted, line-by-line with the line number as the nonce for encryption. We pass lines of the log into the enclave one line at-a-time, because there is no natural division point in the code that implements batching. In future work, one could optimize this code by batching the inputs to the mapper.
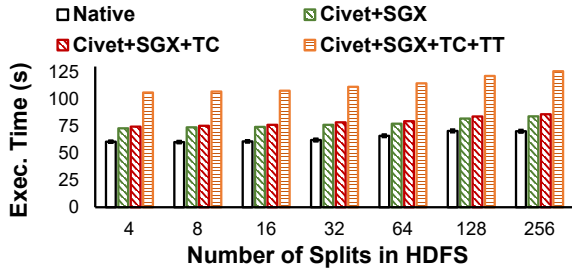
Figure 11: End-to-end execution time of the Hadoop regular expression parser to process 1GB of encrypted authentication logs. Lower is better. For Civet, only the mapper is partitioned into enclaves. We evaluate Civet performance with SGX, deep input type checks (TC), and taint-tracking without explicit flow (TT). The Civet and native workloads both run on a single-node, full-featured Hadoop v2 framework.

Selected entry methods	Shredding	#C	#M	LoC	Δ%
Before partitioning		34.5K	276.9K	3.6M	
`HttpResponder.*`	class	4.2K	37.9K	508.3K	77%
	method	2.0K	11.4K	240.9K	93%

Table 4: Partitioning results for Tomcat, measured in classes (#C), methods (#M), and lines of code (LoC). `RSACipher` and `RSAKeyPairGenerator` are expliclity included for dynamic loading.

However, this has little impact on execution time because our design does not synchronously context switch between enclave and non-enclave execution; rather, Civet follows an exitless pattern. There is a cost of additional CPU cycles (off the critical path) to this design, which batching could reduce.

Hadoop determines the number of mappers and reducers for a given workload based on how many "splits" the data is divided into inside HDFS. We experiment with split sizes ranging from 256MB to 4MB. We observe that, as the number of splits increases well beyond the number of actual cores, the overhead of scheduling degrades performance more than any SGX-specific factor. Civet adds only 16–22% to the end-to-end latency when running with SGX and deep input type checks but without taint-tracking. The overhead of type checks is marginal because of the integration with the class instantiation of Fast-serialization. If taint-tracking is enabled with only explicit flow tracking, the overhead is 70–80%. Furthermore, running a Hadoop task partitioned with Civet is generally as scalable as native.

9.2 Tomcat

Tomcat [63] is a web server for hosting Java servlets in a multi-tenant environment. A servlet is usually written to parse HTTP requests, and can be a building block for microservices. We partition an "echo" servlet into an enclave, which signs
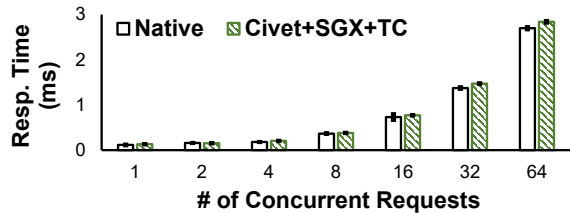

Figure 12: Average HTTP response time of a request-signing Tomcat servlet partitioned and executed by Civet, with SGX and shielded by type checks (TC), compared to native. Lower is better. The HTTP requests are issued by `ab` (ApacheBench), with HTTP request concurrency up to 64.

the HTTP requests from the users using RSA and returns a certificate in the response. This is another good fit for Civet, because the servlet needs to access a secret key to sign the certificate. Thus, tenants do not need to expose their secret keys to the web server or other servlets. Table 4 shows the partitioning efficiency for Tomcat.

Figure 12 shows the average latency to sign requests in a servlet, as a function of the number of concurrent requests. In the Tomcat use case, we observe that the overhead of introducing an enclave in Civet is nearly negligible. The overheads are not SGX-specific, and can be improved by selecting a more scalable configuration for Tomcat.

9.3 GhaphChi

We use GraphChi [79] as more challenging case to partition. We use the **page rank** program in GraphChi as a running example. GraphChi is an in-memory framework for processing large graphs, by sharding vertex and edge data of a graph. The framework includes extensible interfaces for plugging graph algorithms. The core engine, `GraphChiEngine`, is tuned for parallel computing with multiple threads that reuse the graph data cached in the DRAM. We demonstrate the sensitivity to the effectiveness of partitioning using three case studies shown in Figure 13 and evaluated in Table 5.

The simplest, most coarse-grained choice (①) is partitioning at the main function, `Pagerank.main`. This choice will result in a relatively large TCB and the entire program will run inside the enclave throughout the execution. Although this choice does not provide any benefit of partitioning, Civet can still help identify the required classes and methods, and shrink the class libraries.

A finer-grained choice (②) is to partition at each graph operation, e.g, `Pagerank.update()`. This method updates the global `GraphChiContext` with the pagerank contribution of each vertex. This approach will only process one vertex per enclave transition, and is arguably too fine-grained. Worse yet, the input to `Pagerank.update()` is a ChiVertex object, which only contains a pointer to the data blocks; this will require copying the entire data blocks into the enclave for the pointer to be valid. Although this choice is fine-grained in terms of the TCB, the enclave memory footprint is just
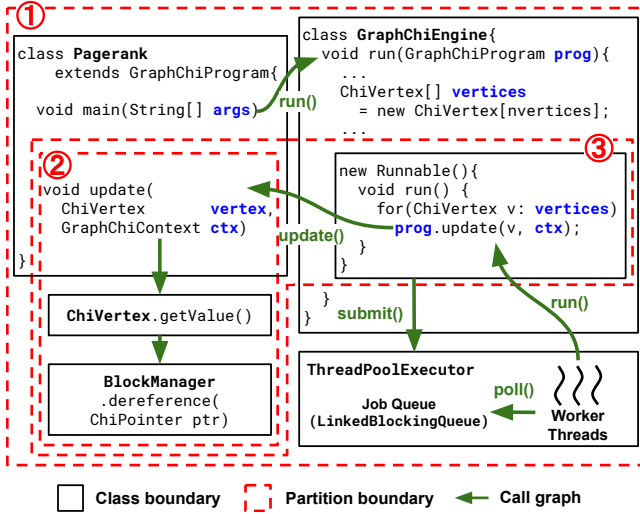
Figure 13: A simplified call graph for the GraphChi page rank program. Execution starts with `Pagerank.main()`, followed by `GraphChiEngine.run()`. GraphChiEngine eventually submits multiple jobs of running the `Pagerank.update()` by the worker threads. We show three possible choices of partition boundary in GraphChi.

Selected entry methods	Shredding	#C	#M	LoC	Δ%
Before partitioning		47.1K	419.5K	4.6M	
① `Pagerank.main`	Class	8.7K	72.5K	1.1M	75%
	Method	3.0K	14.5K	280.2K	94%
② `Pagerank.update`	Class	8.7K	72.5K	1.1M	75%
	Method	2.3K	12.2K	250.2K	95%
③ `GraphChiEngine$2.*`	Class	8.7K	72.5K	1.1M	75%
`GraphChiEngine$3.*`	Method	2.3K	12.2K	250.2K	95%

Table 5: Partitioning results for GraphChi Pagerank, partitioned with three boundaries and measured in classes (#C), methods (#M), and lines of code (LoC). For all three cases, `AESCipher` is explicitly included for dynamic loading.

as large as does not reduce the memory footprint compared to coarser-grained choices. Note that with only class-level shredding, the TCB is the same as ① because the same set of classes are referenced from the entry classes. With method-level shredding, Civet further reduces ∼30K LoC in ②.

A third option (③) is to partition at the granularity of a batch of work, with enough inputs to amortize the enclave transition cost. In the case of GraphChi, chunks of graph data are submitted as `Runnables` to the workers. These `Runnables` are defined as inner classes called `GraphChiEngine$2` and `GraphChiEngine$3`. As shown in Table 5, partitioning at these classes seemingly generates the same TCB as partitioning at `Pagerank.update`, but performs strictly better at run-time.

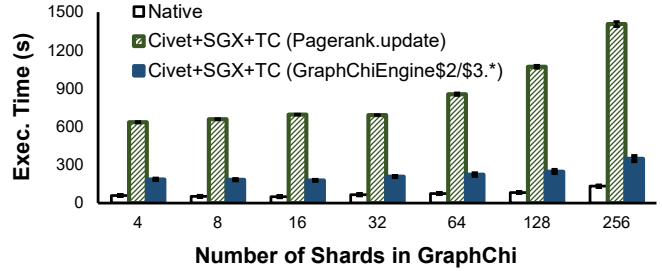Figure 14 shows the execution time processing the page

Figure 14: Execution time of the GraphChi page rank program. For Civet, we tested 2 different choices of partitioning boundary, one with `Pagerank.update` and one with `GraphChiEngine$2/$3.*`. Both encrypt the graph states and are shielded by type checks (TC).

Workloads (Entry methods)	DRAM cost	Processing time			
		Points-to analysis	Shredding	Phosphor	Packaging & signing
(a) Hadoop (②)	4.5 GB	46s	17s	6s	4s
(b) Tomcat (①)	2.5 GB	11s	11s	6s	4s
(c) GraphChi (③)	3.4 GB	21s	11s	6s	4s

Table 6: DRAM cost and processing time (for points-to analysis, shredding, Phosphor instrumentation, packaging, and class signing) of Civet's partition tool. Lower is better.

ranks of the LiveJournal social network [80]. The data set is ∼1.1GB, with 4 million vertices and 69 million edges. Our example shields the partition by encrypting the intermediate graph states (e.g., in and out edges) cached in `ChiVertex` objects. The graph itself is loaded through the file system and can be shielded by the library OS.

We partition the page rank program with the two finer-grained options. We observe that the GraphChi program caches the vertex data and edge data inside the DRAM, using 32768 raw blocks. GraphChi also assigns a memory budget for each job, which decides the range of vertex data to be processed. We reduce the configuration to using 1024 raw blocks and 16MB budget per job, to reduce the memory footprint and RPC overhead. When partitioned with `Pagerank.update`, the overhead can be up to 8.2–12.8× compared to native. Partitioning at `GraphChiEngine$2/$3` lowers the overhead to 1.6–2.5×, due to fewer enclave RPCs.

Performance is generally insensitive to the number of shards, except at very high numbers. Although fewer shards implies fewer RPCs, any savings here are offset by the cost of marshalling a larger data set. Thus, execution time is relatively flat until 64 shards, at which point the cost of additional RPCs dominates and drives up execution time.

	AES	+/-	O	RSA	+/-	O	FFT	+/-	O
Native	.3	.0		475.9	.7		1.6	.1	
Civet	5.3	.0	15.5×	713.8	.7	0.5×	14.8	.1	8.0×
Compute	4.0	.4	11.4×	671.3	.5	0.4×	7.4	.1	3.5×
Input	0.4	.5		19.1	.3		2.8	.0	
Output	1.0	.0		23.3	1.1		4.5	.1	
w/**Phosphor**	8.6	.1	27.7×	1161.5	9.1	1.3×	16.9	.3	9.6×
w/ Implicit flow	22.7	.2	67.8×	4050.6	28.2	7.5×	19.6	.5	11.2×

Table 7: Execution time (in microseconds) of each method and the breakdown of latency in Civet.

9.4 Static Analysis

Table 6 reports the DRAM cost and the processing time for partitioning a Java application. We implement the Civet partitioning tool with Soot 3.3.0 and Apache Byte Code Engineering Library (BCEL) 6.2. Partitioning millions lines of Java code takes up to ~1 minute and 4.5GB of DRAM in our examples. A significant portion of the partitioning time is spent on whole-program points-to analysis. Our Spark configuration includes both application and library classes, and uses on-the-fly call graph analysis and a worklist-based propagation algorithm.

9.5 Microbenchmarks

Table 7 shows the execution time of several microbenchmarks: AES, RSA, and FFT, each of which demonstrate a different performance pattern for partitioned enclave execution. For each of the workloads, we break down the overheads into the computation inside an enclave, and the latency of moving inputs and outputs across the enclave boundary. We note that Native does not incur the cost of moving inputs and outputs.

RSA has the lowest overhead among the three, as the workload is the most computation-intensive. For AES, the inputs and outputs are also small, yet the computation itself suffers up to 11.4× overhead. The difference is that execution outside the enclave can make better use of the AES-NI instructions. FFT demonstrates a relatively data-intensive pattern, and the overhead of transitioning the inputs and outputs is 4.5× in total. Phosphor incurs overhead because of the additional instrumentation and runtime tracing. It performs worst in the AES benchmark (27.7× and 67.8×, without and with implicit flow tracking, respectively), which is the least compute-intensive among the three, showing that the overhead of taint-tracking (with Phosphor) dominates the running time. In contrast, the taint-tracking incurs lower overheads in the more compute-intensive RSA and FFT benchmarks.

9.6 Discussion

The three case studies show the challenges to creating a secure and efficient partition: one must consider not just points to divide the code, but also the data flow and the optimal granularity for moving data in and out of an enclave. Our results show that Civet is very effective at reducing the code footprint for an enclave partition—removing 75% of the code even in the coarsest partition.

In general, Civet introduces an acceptable overhead, end-to-end, for applications. That said, our microbenchmarks indicate up to an 15.5.× overhead on a short computation (AES); thus, optimization such as batching inputs are important to overall performance. Finally, adding dynamic tracking of implicit flows effected by the control flow is considerably more expensive than the rest of Civet. We leave exploration of more efficient implicit flow tracking for future work.

10 Conclusion

This paper presents an enclave-aware JVM variant and a framework for partitioning a large application onto enclaves. Civet leverages language features to help developers reason about the code that is and is not in the enclave. Simply dropping a managed language runtime in SGX incurs an order-of-magnitude slowdown. Civet also minimizes the code footprint in the enclave, as well as adapting the garbage collector to the hardware peculiarities of SGX.

Acknowledgments

References

[1] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

[2] AMD secure encrypted virtualization. https://developer.amd.com/amd-secure-memory-encryption-sme-amd-secure-encrypted-virtualization-sev/.

[3] David Lie, Chandramohan A Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. *ACM SIGOPS Operating Systems Review*, 2003.

[4] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, volume 16, 2016.

[5] 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 *IEEE S&P*, 2015.

[6] 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*, 2017.

[7] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference*, 2016.

[8] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. TrustJS: Trusted client-side execution of JavaScript. In *Proceedings of the 10th European Workshop on Systems Security*, 2017.

[9] Mark Russinovich. Introducing Azure confidential computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, 2017 September.

[10] Pratheek Karnati and Karna Bojjireddy. Data-in-use protection on IBM Cloud – IBM, Intel, and Fortanix partner to keep enterprises secure to the core.

[11] Apache Hadoop. http://hadoop.apache.org/.

[12] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI*, 2014.

[13] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux containers with Intel SGX. In *OSDI*, 2016.

[14] Graphene library OS. http://github.com/oscarlab/graphene.

[15] SGX-LKL. https://github.com/lsds/sgx-lkl.

[16] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library os for unmodified applications on SGX. In *USENIX ATC*, 2017.

[17] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *ASPLOS*, 2014.

[18] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rudiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for Intel SGX. In *USENIX ATC*, 2017.

[19] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *CCS*, 2015.

[20] ARM TrustZone. http://www.arm.com/products/processors/technologies/trustzone/.

[21] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *ASPLOS*, 2013.

[22] CWE-843: Access of resource using incompatible type ('type confusion'). https://cwe.mitre.org/data/definitions/843.html.

[23] Gang Tan and Jason Croft. An empirical security study of the native code in the JDK. In *USENIX Security*, 2008.

[24] Phosphor: Dynamic taint tracking for the JVM. https://github.com/Programming-Systems-Lab/phosphor.

[25] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless OS services for SGX enclaves. In *EuroSys*, 2017.

[26] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *NDSS*, 2017.

[27] Software Guard Extenstions (SGX) SDK for Linux.

[28] sgx-utils. https://github.com/jethrogb/sgx-utils.

[29] Rust SGX SDK. https://github.com/baidu/rust-sgx-sdk.

[30] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of Android applications for trusted execution environments.

In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016.

[31] Adrien Ghosn, James R. Larus, and Edouard Bugnion. Secured routines: Language-based construction of trusted execution environments. In *USENIX ATC*, 2019.

[32] Stefan Brenner, Tobias Hundt, Giovanni Mazzeo, and Rüdiger Kapitza. Secure cloud micro services using Intel SGX. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, 2017.

[33] Marcela S Melara, Michael J Freedman, and Mic Bowman. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments. *arXiv preprint arXiv:1907.13245*, 2019.

[34] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A bytecode translator for distributed execution of "legacy" Java software. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001.

[35] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002.

[36] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *SOSP*, 2001.

[37] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *SOSP*, 2007.

[38] M Miller. Robust composition: Towards a unified approach to access control and concurrency control 2006. *Johns Hopkins: Baltimore, MD*, page 302, 2006.

[39] Adrian Mettler, David A. Wagner, and Tyler Close. Joe-E: A security-oriented subset of java. In *NDSS*, 2010.

[40] Fred Spiessens and Peter Van Roy. The oz-e project: Design guidelines for a secure multiparadigm programming language. In *International Conference on Multiparadigm Programming in Mozart/OZ*, 2004.

[41] Marc Stiegler and Mark Miller. How emily tamed the caml. *Hewlett Packard Labs Tech Report*, 2006.

[42] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *USENIX Security*, 2016.

[43] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

[44] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, 2017.

[45] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[46] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017.

[47] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, 2017.

[48] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, 2017.

[49] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE S&P*, 2018.

[50] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[51] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Euro S&P*, 2017.

[52] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. DroidRA: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.

[53] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[54] Paulo Barros, Rene Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo dAmorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[55] B. G. Ryder. Constructing the call graph of a program. *IEEE Transaction of Software Engineering.*, May 1979.

[56] Mark Weiser. Program slicing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 1981.

[57] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Johns Hopkins University, 1994.

[58] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[59] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI*, 2000.

[60] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[61] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction*, 2003.

[62] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *PLDI*, 1994.

[63] Apache Tomcat. http://tomcat.apache.org/.

[64] AppArmor. http://wiki.apparmor.net/.

[65] Cynthia Dwork. Differential privacy. In *Proceedings of the 33rd international conference on Automata, Languages and Programming-Volume Part II*. Springer-Verlag, 2006.

[66] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010.

[67] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005.

[68] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007.

[69] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, 2006.

[70] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications*, 2006.

[71] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, 2007.

[72] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for real-time privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 2014.

[73] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature—generation of exploits on commodity software. In *NDSS*, 2005.

[74] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *ACM SIGPLAN Notices*. ACM, 2014.

[75] Java garbage collection basics. http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html.

[76] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *ASPLOS*, 2018.

[77] FST: fast java serialization drop in-replacement. https://github.com/RuedigerMoeller/fast-serialization.

[78] Intel® Software Guard Extensions for Linux* OS - SGX driver. http://github.com/01org/linux-sgx-driver.

[79] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[80] LiveJournal social network dataset. https://snap.stanford.edu/data/soc-LiveJournal1.html.