Data flow analysis for Uranus applications

Chan Kwan Yin (3035466978)

December 31, 2020

Abstract

Trusted Execution Environments (TEE) protect applications from privileged attacks running on untrusted systems such as public clouds, but partitioning enclave boundaries is not always a trivial task. Partitions too small would leak data to the untrusted host system, while partitions too huge would result in unnecessarily large trusted computing base (TCB) that increases the risk of overflowing Enclave Page Cache (EPC). A passive analysis approach can be adopted where users annotate data as sensitive sources or sinks, and an analysis tool determines variables considered sensitive and compares it with the enclave boundaries declared.

This project introduces *enclavlow*, an information flow analysis tool for JVM-based projects using Intel SGX enclaves with the Uranus [1] framework. It implements a set of security policies tailored for Uranus-based applications, and reports leaking variables or functions that could be run out of enclave. The analysis tool is delivered as a Gradle plugin to be deployed as a continuous integration tool in Gradle-based projects. The source code for *enclavlow* is released on `https://github.com/SOF3/enclavlow`. The project work can be incorporated into Uranus in the future for optimizations.

[1] Uranus: Simple, Efficient SGX Programming and its Applications. `https://doi.org/10.1145/3320269.3384763`

Contents

List of Figures

List of Tables

Listings

Abbreviations

3AC	Three-Address Code
AFG	Aggregate Flow Graph
API	Application Programming Interface
AWS	Amazon Web Services
CFG	Contract Flow Graph
DFS	depth-first search
DTA	Dynamic Taint Analysis
EPC	Enclave Page Cache
JIT	Just-In-Iime Compiler
JLS	Java Language Specification
JNI	Java Native Interface
JVM	Java Virtual Machine
LCA	Lowest Common Ancestor
LFG	Local Flow Graph
OOP	Object-Oriented Programming
SGX	Software Guard Extension
TEE	Trusted Execution Environment

1 Introduction

1.1 Background

With the rise of third-party public cloud services such as Amazon Web Services (AWS) [1] and Microsoft Azure [7], there is increasing demand for trusted execution where applications are protected from attackers with privileged access to the hardware or software. Modern hardware offer Trusted Execution Environment (TEE) technologies, such as Software Guard Extension (SGX) in Intel CPUs, with which trusted execution code and sensitive data are processed in secure “enclaves”, protected at hardware level to prevent access from other hardware or software layers.

One significant application of TEE is the big data industry, in which confidential user data are processed on cloud servers, and protection from cloud providers may be necessary for compliance with privacy regulations such as GDPR [9]. However, a significant amount of big data libraries and frameworks are written using languages that use Java Virtual Machine (JVM) as the runtime, such as Hadoop [2] and Spark [3]. Recently, Uranus [14], a system for writing SGX applications in Java languages, was released. It provides a simple interface for interacting with SGX in Java, where users annotate methods with `@JECall` and `@JOCall` to move control flow into or out of enclaves. It is the responsibility of the user to determine the correct positions for the `@JECall` and `@JOCall` annotations, namely the enclave boundary partitioning. Since JVM involves a very different approach compared to native applications in terms of software development, distribution and execution, tools designed for native applications are mostly incompatible in JVM.

One important question in enclave programming is the choice of an appropriate enclave boundary. Running the whole application within an SGX enclave is undesirable for two reasons. First, this violates the Principle of Least Privilege, where the whole application becomes possible attack surface for adversaries to compromise protected data [15]. Second, this implies all memory used by the application is placed in the enclave memory (the Enclave Page Cache (EPC)), which is restricted to 100 MB, the overflow of which results in swap memory mechanism, leading to significant performance degrade (“1,000X slowdown compared to regular OS paging” [14]). On the other hand, if the enclave is smaller than necessary, adversaries can either obtain sensitive data directly or infer sensitive characteristics of them indirectly. An enclave boundary is to be selected with high precision to avoid impacts from either side. Ideally, the enclave boundary shall cover the minimum subset of code in which sensitive data are processed such that use of EPC is minimized without resulting in any sensitive data leak.

This project presents *enclavlow* [2], an information flow analysis tool for identifying data leak from enclaves. The user first wraps sensitive data sources in `sourceMarker` and sinks in `sinkMarker`. The tool performs information flow analysis from `sourceMarker`

[2] “enclavlow” is a term coined from the words “enclave” and “flow”.

variables, identifying the ways that data from such variables are leaked to the system outside the executing enclave without first passing through a `sinkMarker` variable. The tool compiles a report in HTML format that lists the flow of sensitive data leak from the source marker to areas accessible by privileged adversaries.

enclavlow is shipped as a Gradle plugin, providing a Gradle task that takes the `*.class` files compiled in the `classes` task and generates the report for the analysis from those classes.

1.2 Prior work

Information flow analysis is not a new technology in the field. While this project analyzes JVM code using SGX enclaves, prior research on *automatic partition* and non-SGX dynamic analysis was found.

Glamdring [15] is a C framework that automatically selects the minimal SGX enclave boundaries based on user requirements specified through C pragma directives. However, since the process is fully automated, it has a lower tolerance of false positives, which increases the risk of unintentional data leak. This project, unlike Glamdring, will only perform analysis but not automatic partitioning, allowing for greater false positive tolerance.

Phosphor [10] is a Dynamic Taint Analysis (DTA) framework that modifies Java bytecode to add tags to sensitive data at runtime and check if such tags are leaked. Although dynamic taint is more accurate, this project prefers a static analysis approach, which enables developers to identify sensitive regions at compile time without the need to feed concrete data into methods.

Civet [11] is a framework for Java code partitioning. Similar to Glamdring, it automatically selects the minimal enclave boundaries. It is also similar to the goal of this project, except that Uranus provides additional protection at the native level, allowing further optimizations compared to Civet. Challenges found in this project, such as polymorphism complexity, were also observed with Civet.

This report will focus on the algorithm used to detect data leaks, as well as performance concerns and limitations from the JVM design. An example application is evaluated to demonstrate the functionality of *enclavlow*.

2 Objectives

This section describes the usage and intended behaviour of *enclavlow*.

2.1 Marker API

The `enclavlow-api` Gradle submodule in the project is a library exposing two identity functions as expression markers as shown in Listing 1. Corresponding definitions such as `intSourceMarker` are also defined for the eight Java primitive types for both source and sink (omitted in Listing 1 for brevity). Users can wrap *ultimate* sensitive data sources with `sourceMarker` and mark *intended* leaks with `sinkMarker`. It is expected that the identity wrapper function is optimized away by Just-In-Iime Compiler (JIT).

Listing 1: Definition of `sourceMarker` and `sinkMarker`

```
package io.github.sof3.enclavlow.api;

public class Enclavlow {
    public static <T> T sourceMarker(value: T) { return value; }

    public static <T> T sinkMarker(value: T) { return value; }
}
```

Exceptions induced from the expression of parameters to `sinkMarker` are not collected by the sink. This is because throwing sensitive data is a rare use case, has a wide range of scenarios and highly depends on the exact class thrown. If sensitive information in an exception is intended for leaking, a more verbose catch-wrap-throw syntax can be used as demonstrated in Listing 2. `sinkMarker` can also be used to explicitly suppress false positives generated by *enclavlow*.

Listing 2: Catch-wrap-throw construct

```
try {
    thisMethodThrowsSensitiveExceptions();
} catch(SensitiveException e) {
    throw sinkMarker(e);
}
```

A simple example usage is shown in Listing 3. In particular:

- On line 4, `raw` is not `sourceMarker` because parsing is a late stage after raw data extraction and `sourceMarker` should only be applied on the ultimate source, which is asserted on line 5 that “computing the sum of `raw` is a legitimate leak”.
- `result` on line 14 is not marked `sinkMarker` because the leak of sensitive information should be analyzed. `sum` on line 22 is not marked `sinkMarker` because it is a sensitivity-neutral utility function that does not imply any assertion on whether the leak is acceptable; otherwise incorrect behaviour is created by changing line 12 to Listing 4, where `parse` no longer returns a security-sensitive value.

Listing 3: Simple example of `sourceMarker` and `sinkMarker`

```
class SourceSinkExample {
    @JECall
    int getSum(byte[] encrypted) {
        List<Integer> raw = parse(encrypted);
        return sinkMarker(computeSum(raw));
    }

    List<Integer> parse(byte[] encrypted) {
        byte[] buf = sourceMarker(PRIVATE_KEY.decrypt(encrypted));
        List<Integer> result = new ArrayList<>();
        for(byte i : buf) {
            result.add((int) i);
        }
        return result;
    }

    int computeSum(List<Integer> integers) {
        int sum = 0;
        for(int i : integers) {
            sum += i;
        }
        return sum;
    }
}
```

Listing 4: Example of leak through `computeSum`

```
result.add(computeSum(Collections.singletonList((int) i)));
```

2.2 User interface

The `enclavlow-plugin` Gradle submodule is a Gradle plugin providing a task `:enclavlow`, which depends on the `:classes` builtin task and performs flow analysis on the class binaries. The analysis report is generated as HTML format in the Gradle reports directory (build/reports/enclavlow.html). The report contains the following elements:

Flow summary Each method that sensitive data flow through are reported. If multiple invocations lead to different data flows, the union of such data flows is displayed.

Data leaks Code that which results in immediate data leaks, such as methods passing security-sensitive data to other `@JOCall` methods or methods marked `@JECall` returning/throwing security-sensitive data, are highlighted in an index called "Sensitive leaks". For each highlighted method, the developer should move it into enclave boundaries, or adjust the `sourceMarker` / `sinkMarker` wrappers.

Contract graphs The contract graph of each method is provided as hyperlinks in the HTML report and can be viewed using the Graphviz Online tool [5].

When debug mode is enabled, the LFG (see section 3.1.2) of each method can also be viewed in DOT/SVG format from `lfgOutput` under the Gradle build directory.

2.3 Threat model

The adversary of concern is expected privileged access to the host system, including other threads in the JVM runtime, the JVM runtime itself, other (root) processes, the operating system kernel, the hypervisor and the BIOS. The adversary cannot access the SGX execution part of the CPU as cryptographically provable by SGX attestation mechanism. While the realistic adversary also has access to hardware resources such as system clock and temperature sensor modules, which can be used to implement side-channel attacks, analyzing such side channels involves system-, environment- and architecture-dependent analysis. As these side channels rarely expose meaningful and specific information, they are beyond the scope of this project.

The security model provided by Uranus is also slightly altered. Uranus rejects assignment to objects outside enclaves to mitigate some attacks like the Iago attack. On the other hand, *enclavlow* considers such writes to be rvalue-leaking and control-flow-leaking, but still assumes them permissible. This change intends to target future changes in Uranus that perform compile-time flow analysis.

3 Methodology

The main component of this project is the analysis framework, which is conducted in the form of iterations to fulfill security policies as specified in the integration tests. Other parts such as Gradle plugin interfae, although necessary for usage, are not focus areas of this project, and hence will not be discussed further.

3.1 Design

Since the adversary in the threat model has arbitrary access to any untrusted memory and instruction, the security policies of *enclavlow* differ slightly from typical information flow analysis. For instance, the statement `a.b.c = d;` usually does not propagate the effect of `d` to `a`, but since the adversary is capable of changing `b` to any memory location of its favour, the security of this statement depends on whether `a` is trusted.

enclavlow adopts a flow graph approach, where each node represents an element that may be leaked.

Table 1: Nodes in CFG

Static	Data located in static class fields
Param x	Each parameter corresponds to a Param node
This	The object on which a method was invoked
Return	Information flow through the throw path
Throw	Information flow through the return path
Source	`sourceMarker` values
Sink	`sinkMarker` values
Control	See section 3.1.3
Proxy nodes	Param y, Return, Throw and Control for each function call

Definition 1. Given a flow graph (V, E), for all $x, y \in V$, $(x, y) \in E$ if and only if allocating y outside an enclave reduces the indistinguishability of x even if all other nodes are removed.

If the "even if" condition is removed, this is a transitive relation. The edges $\{(x, y), (x, z)\} \subset E$ imply that leaking *either y or z* already allows the adversary to distinguish x. However, there is no way to represent that x is distinguishable if *both y and z* are distinguished. This limitation is resolved by creating additional nodes that delegate x.

Note that this definition ("our definition") differs slightly from the usual definition of flow graphs (especially in DTA), where an edge (x, y) represents the flow of information from x to y [18]. In the usual definition, only read access for the adversary to y is concerned, while in our definition, the adversary possesses write access to x.

enclavlow constructs flow graphs in three stages to reduce local opcodes to a cross-method flow graph. Each method is first analyzed independently into a LFG. The LFG is then contracted into a subgraph of only non-local nodes, namely the CFG. The CFG of each method is merged together to form the Aggregate Flow Graph (AFG), on which enclave boundary intersections are identified to be leaks.

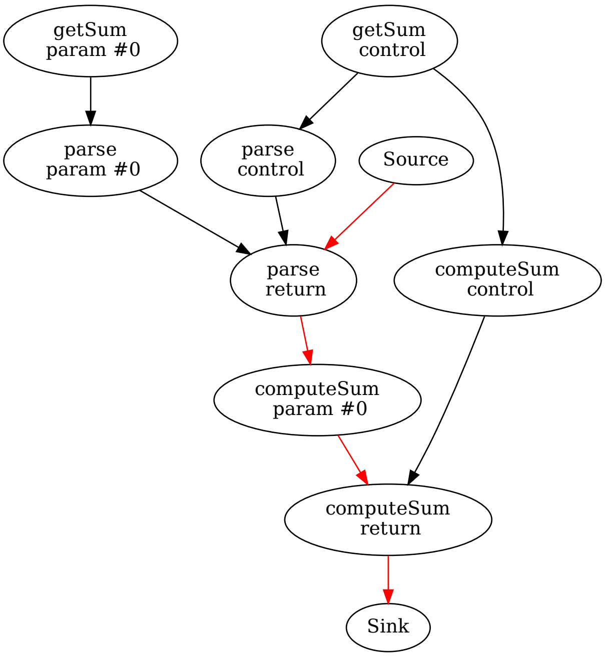
3.1.1 Contract Flow Graph (CFG)

enclavlow constructs a CFG for each method analyzed. The CFG contains the nodes listed in Table 1. In this article, the names for flow graph nodes are underlined.

The CFG is computed by contracting the LFG, as described in the section 3.1.2. After all methods in a class are evaluated, the CFGs of child methods called from the analyzed methods are lazily evaluated as well. All CFGs are merged into an AFG, joined using the function call nodes.

Figure 1 shows an example of AFG. Unconnected nodes are omitted for brevity.

Figure 1: Example CFG for Listing 3



3.1.2 Local Flow Graph (LFG)

To construct the CFG, a local flow graph is constructed to identify the information flow between temporary variables. Using `ForwardBranchedFlowAnalysis` from the Soot [12] framework, the analysis follows along the control flow of the program. Java bytecode of the program to analyze is converted into Jimple code, a Three-Address Code (3AC) language. Each 3AC statement maps a LFG to another LFG, with the exception of *branch* operations (`if` / `switch`) that map one flow graph to multiple branches and *merge* operations that map multiple branches to one. For the case of loops, the analysis repeats until the LFG converges.

The LFG extends the CFG with the following additions:

- Each local variable (some may only exist as intermediate values in source code) is allocated a node.
- Each branch has its own Control node.

For assignment statements, rvalue nodes for the source, side effect nodes from the destination and the current control flow contribute edges towards the lvalue nodes for the destination. The rvalue and lvalue nodes depend on the expression type:

Literals As rvalues, these values do not leak any information. They cannot appear as lvalues.

Local variables Local variables may be declared or temporary expressions. As mentioned above, each local variable has its own node.

Parameters and this Jimple always assigns parameters to a local variable first, so they can only appear as rvalues. The only usage for parameters and this in Jimple code is some statement like `r0 := @this:  IagoAttack;`,

Object field references A new *projection* node is created for the field if it does not already exist. An edge from the field to the object (`FIELD_PROJECTION_BACK_FLOW`) is added. For each assignment with the object value, assignment edges are created from the equivalent fields to the new field.

For the case of static fields, reads are always considered to have no leaks, while writes are always considered to be leaks as mentioned in the threat model section above.

Array references `a[b]` is treated as a special field of `a`. If used as an lvalue or an rvalue, `b` also contributes as rvalue.

Array literal The array size contributes as an rvalue as it can be directly recovered from the array. Since mutations have no effect on the size value provided in the array literal, this is effectively the same as `array.size = inputSize;`.

Type casts, instanceof expressions and other operators As this only affects at the type level, type casts and instanceof have the same semantics as their value operands. Unary and binary operations have the same behaviour as the union of rvalue nodes from their operands.

Method calls Calling methods creates blackbox proxy nodes to the nodes in their CFG, with the exception of `sourceMarker` and `sinkMarker`, which are resolved as Source and Sink directly. The Sink node is actually not necessary for analysis, but is cosmetically helpful for interpreting sensitive data flow in the output.

For most expression types above, corresponding test cases can be found on the GitHub repository under the directory `core/src/test`.

Since the mapping of flow graphs is deterministic, overwriting a variable does not remove its body.

At exit points, the CFG is constructed by considering the contract nodes that flow into This, Static, Sink, all Params and all blackbox proxy nodes. For return and throw statements, the flow into the returned/thrown value is also considered.

3.1.3 Handling control flow

Sensitive data can be leaked through control flow, such as in Listing 5. The code effectively copies the value of `secret` to `result`. To handle this case, a special node called Control is introduced to denote the information that can be revealed through the control flow.

Listing 5: Leak by control flow

```
int controlFlowLeak(int secret) {
    int result = 0;
    for(int i = 0; i < secret; i++) {
        result++;
    }
    return result;
}
```

The control node CTRL is defined with these semantics:

Definition 2. For a local flow graph (V, E),

- $(x, \text{CTRL}) \in E$ implies that there exists a deterministic predicate p such that some code represented by CTRL is executed if $p(x)$ is true.
- $(\text{CTRL}_1, \text{CTRL}_2) \in E$ implies that CTRL_2 is a branch that executes only if CTRL_1.
- $(\text{CTRL}, y) \in E$ implies that leaking y allows the adversary to distinguish whether CTRL was executed.

When branch statements are encountered, a new Control node is created for each branch, with an edge from the old node. When the control flow converges, the Lowest Common Ancestor (LCA) of the Control of each input LFG is selected as the new Control.

Each method call also involves its base control node; for example, a `@JOCall` method always leaks everything including its control flow. An empty `@JOCall` method leaks exactly just its own control, so it is necessary to expose method control nodes as part of the contract graph.

3.1.4 Aggregation

The AFG simply merges all CFGs by linking proxy nodes together. The presence of sensitive information leak is detected by searching one of the following types of edges in a depth-first search (DFS) from Source:

- Return/throw paths of `@JECall`s
- Parameters/call context of `@JOCall`s
- Control flow of `@JOCall`s
- Assignment to Static

3.2 Implementation details

Local and contract flow graphs are considered "dense" graphs, and are represented using edge matrix. Meanwhile, the aggregate flow graph has a much lower density of edges, and an edge list is much more memory-efficient.

Since Soot uses LFG equality as the predicate to decide whether loops have to be analyzed again, to avoid infinite loops, Control nodes and function calls are excluded from consideration when comparing the graph.

3.3 System Requirements

The logical code of this project is mostly implemented in Kotlin, a JVM language with more concise syntax than Java. However, to ensure that the behaviour analyzed is as explicit as possible, all test cases are written in Java.

As Uranus was only tested against Linux systems, this project does not intend to support other operating systems. Furthermore, due to challenges with classpath detection, only OpenJDK Versions 8 and 11 are supported currently. Nevertheless, since *enclavlow* is just a developer tool, its runtime is actually independent of that targeted by Uranus, so it is possible to test support for other platforms in the future.

enclavlow is packaged as a Gradle plugin, allowing developers to use it in projects with a Gradle toolchain. However, the `enclavlow-core` subproject can be reused in other contexts, such as Maven plugins, IDE plugins, etc. With Kotlin toolchain integrated, it is possible to include enclavlow code into Uranus for compile-time analysis.

3.4 Flow analysis framework

Soot [12] was selected as the framework for conducting flow analysis. Although multiple flow analysis systems using Soot already exist, they are not designed against SGX enclave protection. *enclavlow* adopts more strict security policies to prevent attacks from more privileged attackers, unlike traditional information flow analysis that mostly detects user input as the source of insecurity.

Soot accepts Java bytecode files (`*.class`) as input and compiles them into Jimple 3AC, so inputs compiled from different JVM languages are still compatible. Nevertheless, some languages generate a lot of boilerplate code, such as null assertion code from Kotlin, which reduces readability of the report.

Other flow analysis frameworks were also considered, such as Joana [6] and JFlow [16]. Soot was chosen due to its distinctively thorough documentation and builtin support for call graph analysis.

3.5 Testing

This project uses JUnit 5 and `kotlin.test` framework to conduct unit tests. Test case classes are compiled together in the `:core:testClasses` task, which declares dependency on the APIs `sourceMarker` and `sinkMarker`, but this is not necessary for actual usage.

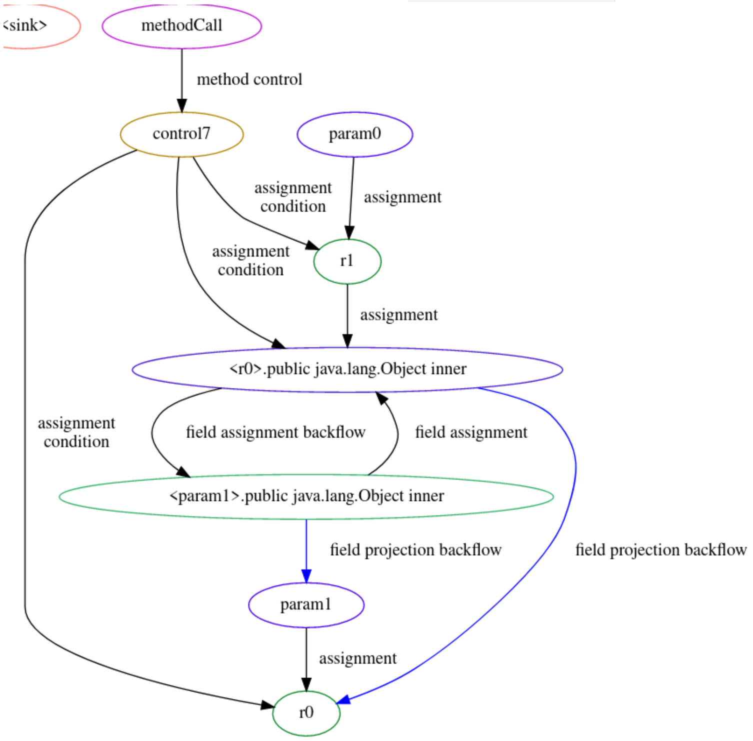
4 Evaluation

For evaluation, the project conducts unit testing to verify correctness of each case of information leak in concern, and integration testing with a profiler to evaluate performance.

4.1 Correctness

The following test cases are a subset of unit tests taken from the `core/src/test` directory on the GitHub repository.

Figure 2: LFG for `paramToParam`



4.1.1 Field projection

First consider field projection tests in Listing 6. The `paramToParam` method assigns a value to a reference. Hence, their contract graphs shall have the method Control as well as the assigned Param 0 flow to the field of the destination, as seen in the LFG in Figure 2.

Listing 6: Projection test case

```java
    int b;

    public void paramToThis(int x) {
        b = x;
    }
```

Note the mutual flow between the nodes `<r0>.public java.lang.Object.inner` and `<param1>.public java.lang.Object.inner`. Since `param1` is assigned into the local temporary `r0`, they point to the same value, so their common field `inner` are located at the same address, hence flow to one node always flows to the other.

4.1.2 Control flow

Branches and loops involve more Control nodes in the LFG.

Consider the loop tests in Listing 7. The `loopDec` method effectively copies `i` to `a`, leaking the parameter to return value by control flow. The Control from branching by `i` covers the assignment of `a`, allowing it to reach the return path. Similarly, the `whileCall` method uses the return proxy node of `supplier.getAsBoolean()` to branch, leaking any possible returned flow from `getAsBoolean()` to its own return path.

Listing 7: Loop test cases

```java
public static int loopDec(int i) {
    int a = 0;
    while (i-- > 0) {
        a += i;
    }
    return a;
}

public static int whileCall(BooleanSupplier supplier) {
    int i = 0;
    while(supplier.getAsBoolean()){
        i++;
    }
    return i;
}
```

The test results can be reproduced with the Gradle test task. Consult the GitHub CI setup at `https://github.com/SOF3/enclavlow/actions` for details on environment setup. Other unit tests in the project are omitted in this report as their functionalities are insignificant or already covered by the above cases.

4.2 Performance

To analyze the performance of *enclavlow* in real projects, a test application is written to run the *enclavlow* gradle plugin with. The source code is available under the `example/` directory on the GitHub repository. The test application opens a UDP socket that listens for packets and computes a 6-digit one-time-password from the packet data.

The application involves the 6 simple methods of its own, as well as the following methods from Java standard library:

- `DatagramPacket.getData()`
- `DatagramPacket.new(byte[], int)`
- `DatagramPacket.setData(byte[])`
- `DatagramSocket.bind(InetSocketAddress)`

- `DatagramSocket.isClosed()`
- `DatagramSocket.new()`
- `DatagramSocket.receive(DatagramPacket)`
- `DatagramSocket.send(DatagramPacket)`
- `InetSocketAddress.new(String, int)`
- `System.currentTimeMillis()`

Due to transitive method calls, it is found that 1806 different methods were eventually included in the analysis (including native and abstract methods, which are not handled correctly; see section 5.1 for details). A total of 86 seconds were spent for analysis on an Intel i7-10510U CPU (1.80GHz) (number of cores does not matter since the program is largely single-threaded) on Linux kernel 5.4.0-58-generic with Java profiler enabled.

Profiler result shows that, among the 56 seconds of CPU time used on running Java code (the remaining is attributed to JVM native operations), 76.8% time is spent on `soot.SootResolver.resolveClass`, which is the part where Soot compiles the Java bytecode into its own representation. Only 18.7% time was actually spent on code from this project, i.e. the part performing flow analysis.

Since Soot is not thread-safe due to its unfortunately aggressive use of static variables, it is not possible to improve performance using parallelism. Major change on the Soot framework or JVM forking is necessary for performance improvement.

Soot issues aside, a large amount of CPU time is spent on repacking the LFG due to node removal. A more efficient graph representation structure (such as layered views) may be helpful in improving performance in this part.

A full profiler report can be found at `docs/profiler-results.jfr` on the GitHub repository. Figures 3 and 4 show the flame graph of profiler samples.

5 Discussion

There are various unimplemented features in this project, which can be tackled in future research.

Figure 3: Profiler flame graph (full)

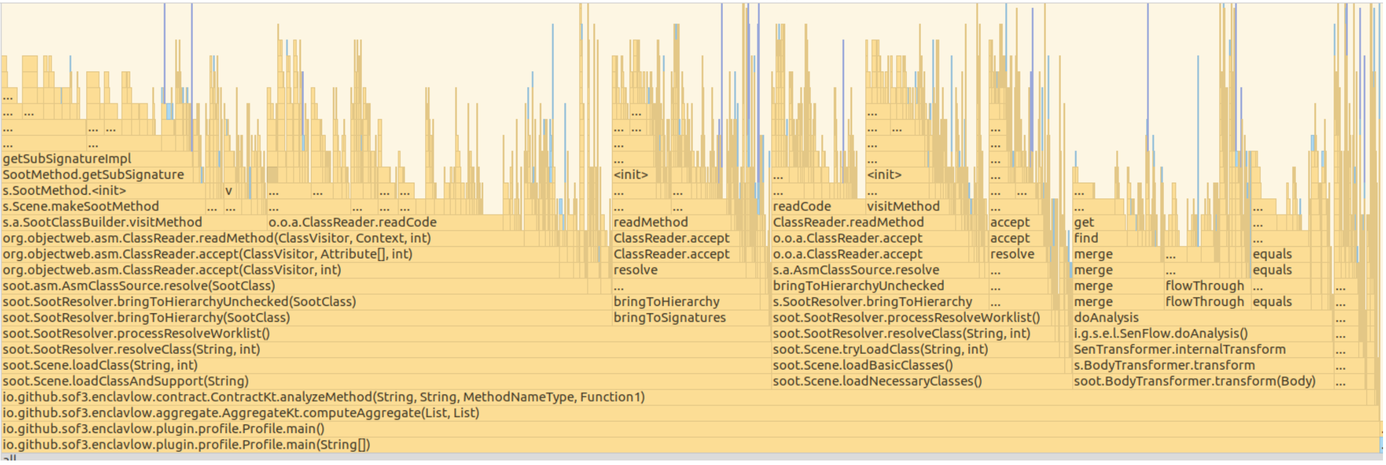
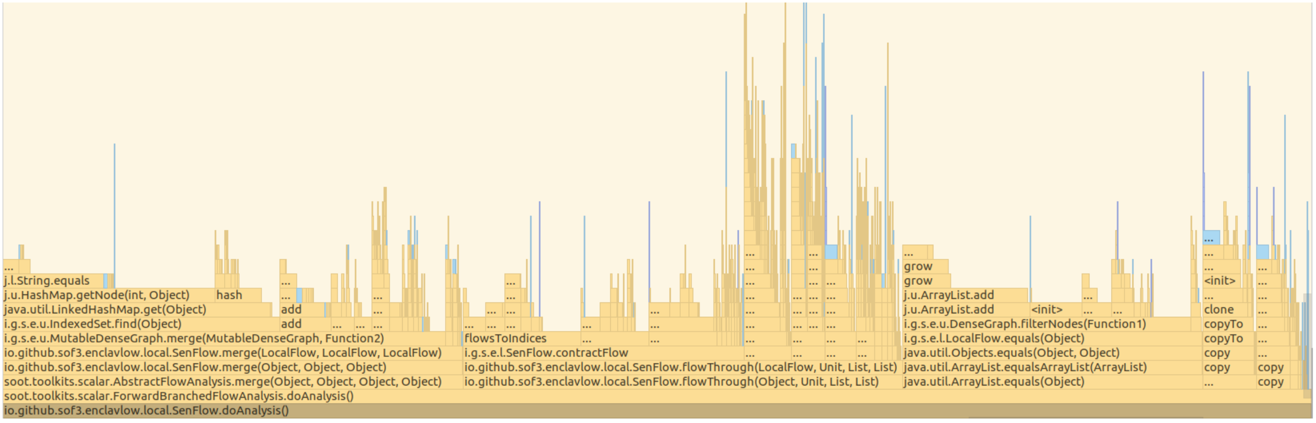


Figure 4: Profiler flame graph (analysis only)



5.1 Limitations

As is every code analysis system, perfect detection is almost impossible. This subsection will show that the requirements of *enclavlow* is a *superset* of the common analysis tools in terms of problems to identify.

5.1.1 Polymorphism

The principles of Object-Oriented Programming (OOP) imply that the receiver of a method call may be swapped with a compatible implementation in another subclass that performs different actions than the current one. Although Uranus prevents the adversary from passing arbitrary malicious code into the enclave memory, it is still possible to pass objects of unexpected but trusted subclass through the `@JECall` boundary. Alternatively, an alternative form of Iago attack passes originally impossible combination of subtypes to the function, which could also introduce attack vectors. Consider Listing 8 for example. If `cs` is passed with a `substr` implementation that writes its parameters to a static variable, the function would leak the length of the security-sensitive `secret`, which is not desirable. To correctly solve the vulnerability of OOP substitution, it is necessary to perform call graph analysis on the actual classes passed to the method, which involves more complex framework level work.

Listing 8: Iago attack through substitution

```java
class OopSubstAttack {
    @JECall
    public void foo(CharSequence cs) {
        byte[] secret = sourceMarker(new byte[0]});
        writeEncrypted(cs.substr(secret.length()));
    }
}
```

Due to time constraints, polymorphism handling is not implemented in this project. Instead, a *degenerate* contract graph with the edge set $\{(\underline{\text{Param}}\ x, \underline{\text{Return}}) : x\}$ is used as placeholder for abstract methods, and possible subclass overrides are not yet considered.

5.1.2 Native methods

Similar to abstract methods, Java Native Interface (JNI) methods are also not easy to analyze. The implementation of JNI methods is only available in the form of native dynamic libraries, and reverse engineering such libraries is unnecessary, tedious and sometimes illegal. A much simpler but accurate approach is to analyze the native methods in the original languages they were written in, integrating with tools such as Glamdring [15]; for the native methods within Java standard library, it is possible to precompute the contract of each of them manually and ship these with the software. Similar to above, this task is omitted due to time constraints; degenerate contracts are used as placeholder instead. This assumption is inaccurate; a common counterexample is the `System.arraycopy` method, which leaks the values of the integer parameters into the destination array parameter since the amount of elements changed leaks the integers involved.

5.1.3 Duplicate code

Despite optimizations and simplifications, it is still not possible to perform perfectly accurate information flow analysis within efficient time complexity [17]. For example, this project merges conditional branches together by taking the union of flow graphs, resulting in easy false positive cases. Listing 9 is an example of this: the return statement appears in both arms, so according to the mechanism explained above, the control flow shall leak the branching condition `secret` to the return path. Nevertheless, but the returned value is in fact always `x`. Since this is a minor use case, this bug is considered not worth fixing. In fact, duplicated code in multiple branch arms is often regarded as an antipattern as well.

Listing 9: False positive by duplicate code

```
class KnownFalsePositives {
    static int foo(int x) {
        boolean secret = sourceMarker(1);
        if (secret) {
            return x;
        } else {
            return x;
        }
    }

    static class Ref<T> {
        T t;
    }
}
```

Similarly, self-anonymized sensitive data are not identified correctly, such as in Listing 10. It is believed that such errors are easy for a human to discover and could be marked `sinkMarker` directly, or simply copying `a` to another variable, so these false positives do not have major effect on usability. Note also that mutably sensitive data may be an antipattern as the user may accidentally leak it in the future by moving a line of code incorrectly.

Listing 10: False positive by self-anonymization

```
@JECall
static int foo(int a) {
    int secret = getSecret();
    a += secret;
    doSomethingWith(a);
    a -= secret;
    return a;
}
```

5.1.4 Implicit exceptions

Apart from explicit `throw` statements, runtime exceptions can also be raised when invalid operations are performed, leaking information through control flow. Consider Listing 11. While the mechanism of *enclavlow* does not discover any bugs, the method throws an ArrayIndexOutOfBoundsException if and only if `a >= index`, so an adversary could pass varying values for `a` to binary-search the value of `index`.

Listing 11: Implicit exception

```java
@JECall
static void implicit(int a) {
    byte[] buffer = new byte[a];
    new Random().nextBytes(buffer);

    int index = sourceMarker(5);
    byte selection = buffer[index];

    saveData(sinkMarker(selection));
}
```

This is difficult to notice even for an alert developer. Solutions are unfortunately usually achieved at the source code level, such as using annotations like `@NotNull` and `@Size`. As this project does not intend to reinvent the wheel, prevention of such vulnerabilities is left as the task for other source-level tools. In fact, such exceptions tend to cause other security issues in addition to enclave security, well known as the "billion dollar problem" [13]. It is likely more efficient to try to avoid such unexpected behaviour by fusing the source code before it gets compiled.

5.1.5 Readability

For the sake of consistency with Uranus, it was originally intended to expose `sourceMarker` and `sinkMarker` as annotations on local variables instead of method calls. However, Java Language Specification (JLS) 9.6.1.2 explicitly stated that “an annotation on a local variable declaration is never retained in the binary representation” [8], so a method call based approach is used instead. It is expected that JIT optimization removes the cost involved from an extra method call at JIT compile time.

This also leads to readability issues. The current HTML output is unable to reveal more information than the CFG because of the lack of information about the source code, like local variable names and line numbers. In the LFG, generated variable names like `r0` and control marker names like `control13` are used instead, but they are not intuitive for the reader and become hard to interpret when the method grows large. The only solution for this problem is to provide additions at the source code level, but since *enclavlow* takes inputs in the form of Java class binaries, this is considered out of scope of the project.

5.2 Recommended future research

There are multiple areas in which this project can be extended.

enclavlow applies handwritten heuristics to identify leaks. Some special edges, such as field projection, are not very well-defined. This reduces the reliability of *enclavlow* in terms of robustness in targeted attacks, which is an important feature for security analysis. A formal proof through tools like Coq [4] can be utilized to ensure that the LFG construction does not miss marginal cases.

The project can also be used to improve Uranus performance. Currently, to ensure enclave confidentiality, Uranus requires enclave code accessing untrusted memory to use Uranus's untrusted-memory Application Programming Interface (API) like `SafeGetField` and `SafeWriteField` [14], which checks the memory address against enclave bounds at runtime. The static analysis in *enclavlow* allows Uranus to validate these bounds at compile time, hence avoiding the runtime bounds-checking cost and improve performance. Note that JIT optimization is not able to detect unnecessary bounds checking. Since `enclavlow-core` exposes the AFG result fully, it shall be possible for the compiler in Uranus to directly invoke this library to analyze the program being compiled.

6 Conclusion

This project aims to develop a JVM code analysis tool for software using Uranus for SGX applications to assist the choice of enclave boundaries. A divide-and-conquer approach was adopted for efficient abstraction of information flow across a method. Since the project delivers an analysis tool but leaves the decision right to the user, a higher tolerance for false positives was accepted.

To formalize the behaviour of false positives with the project, analysis is to be conducted on the occurrence of false positives. Usability of the tool with common libraries in the Java ecosystem will be assessed. With sufficient theoretical background to support the correctness of the algorithms, this tool is expected to serve as an auxiliary quality control integration for open source projects that may see demand in the big data industry and other confidential data processing applications.

The source code of this project is released at `https://github.com/SOF3/enclavlow`.

References

[1] Amazon web services. https://aws.amazon.com.

[2] Apache hadoop. https://hadoop.apache.org.

[3] Apache spark. https://spark.apache.org.

[4] The coq proof assistant. https://coq.inria.fr/.

[5] Graphviz online. https://dreampuf.github.io/GraphvizOnline/.

[6] Joana (java object-sensitive analysis). https://pp.ipd.kit.edu/projects/joana/.

[7] Microsoft azure. https://azure.microsoft.com/en-us/.

[8] Java language specification, third edition. https://docs.oracle.com/javase/specs/jls/se6/html/interfaces.html, 2005.

[9] General data protection regulations. https://gdpr-info.eu/, 2016.

[10] Jonathan Bell and Gail Kaiser. Phosphor: illuminating dynamic data flow in commodity jvms. OOPSLA '14, pages 83–101. ACM, 2014.

[11] Chia che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. Civet: An efficient java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 505–522. USENIX Association, August 2020.

[12] Arni Einarsson and Janus Dam Nielsen. A survivor's guide to java program analysis with soot. *BRICS, Department of Computer Science, University of Aarhus, Denmark*, 17, 2008.

[13] Tony Hoare. Null references: The billion dollar mistake. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-

[14] XC Jianyu Jiang, CW Tzs, On Li, T Shen, and S Zhao. Uranus: Simple, efficient sgx programming and its applications [unpublished]. In *Proceedings of the 15th ACM ASIA Conference on Computer and Communications Security (ASIACCS '20)*, 2020.

[15] 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)*, pages 285–298, 2017.

[16] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[17] Geoffrey Smith. Principles of secure information flow analysis. In *Malware Detection*, volume 27 of *Advances in Information Security*, pages 291–307. Springer US, Boston, MA, 2007.

[18] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. CCS '07, pages 116–127. ACM, 2007.