Data flow analysis for Uranus applications

Chan Kwan Yin (3035466978)

28 October 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`.

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

Contents

List of Figures

List of Tables

Listings

1 Abbreviations

3AC Three-address Code

AFG Aggregate Flow Graph

CFG Contract Flow Graph

CLI Command Line Interface

DTA Dynamic Taint Analysis

EPC Enclave Page Cache

GIGO Garbage In, Garbage Out

JLS Java Language Specification

JNI Java Native Interface

JVM Java Virtual Machine

LFG Local Flow Graph

OOP Object-oriented Programming

SGX Software Guard Extension

TEE Trusted Execution Environment

2 Introduction

2.1 Background

With the rise of third-party public cloud servics such as AWS [1] and Microsoft Azure [5], there is increasing demand for trusted execution where applications are protected from attackers with privileged access to the hardware or software. Modern hardware offer TEE technologies, such as SGX in Intel CPUs, with which trusted execution code and sensitive data are processed in secure "enclaves", which is protected at hardware level to prevent access from other hardware or software layers.

One significant application of TEEs is in big data processing, where confidential user data are processed, and protection from cloud providers may be necessary for complianc with privacy regulations such as GDPR [7]. However, a significant subset of such applications are written using languages that use JVM as the runtime, such as Hadoop [2] and Spark [3] Recently, Uranus, a system for writing SGX applications in Java languages, was released [11]. It provides simple interface for SGX, 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, compared to native applications running on the CPU, involves an entirely different approach with regard to software development and distribution, the tools applicable for native applications are mostly incompatible with JVM, introducing the corresponding new research areas.

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 [12]. Second, this implies all memory used by the application are placed in the enclave memory (the EPC), which is restricted to 100 MB before significant performance degrading ("1,000X slowdown compared to regular OS paging") [11]. 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.

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` 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 summarizes the following:

- **Data leak**: The report displays the lines of code on which sensitive data are moved into areas accessible by privileged adversaries. It demonstrates the path from the `sourceMarker` expression to the point of leak.
- **Redundant protection**: The report lists the functions that could not hold any sensitive data in its local variables, hence should be moved out of the enclave partition.

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.

2.2 Prior art

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

Glamdring [12] 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 [8] is a 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 [9]

[2] "enclavlow" is a new term coined from the words "enclave" and "flow".

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

```
1  package io.github.sof3.enclavlow.api;
2
3  public class Enclavlow {
4      public static <T> T sourceMarker(value: T) { return value; }
5
6      public static <T> T sinkMarker(value: T) { return value; }
7  }
```

Listing 2: Catch-wrap-throw construct

```
1  try {
2      thisMethodThrowsSensitiveExceptions();
3  } catch(SensitiveException e) {
4      throw sinkMarker(e);
5  }
```

3 Objectives

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

3.1 Marker API

The `enclavlow-api` Gradle submodule in the project is a `compileOnly` library exposing two annotations as shown in Listing 1. Corresponding definitions such as `intSourceMarker` are also defined for the eight Java primitive types for both source and sink but omitted here.

Users should wrap *ultimate* data source variables as `sourceMarker`, and mark *acceptable* leaks as `sinkMarker`. Note that throwing sensitive data is a rare use case, has a wide range of scenarios and highly depends on the exact class thrown. If an exception is expected to be marked `sinkMarker`, the developer should use a more verbose syntax of `catch`ing the exception, wrapping it with `sinkMarker` and throwing the wrapped expression as in Listing 2.

`sinkMarker` can also be used to explicitly suppress false positives generated by *enclavlow*. A simple example usage is shown in Listing 3.

Attention to be given to the following points:

- On line 4, `raw` is *not* `sourceMarker`. This is because parsing is a late stage after raw data extraction, and `sourceMarker` should only be applied on the ultimate source.
- Line 5 asserts that "computing the sum of `raw` is a legitimate leak".
- On line 14, `result` is not marked `sinkMarker`, because the leak of sensitive information should be analyzed.
- On line 22, `sum` is not marked `sinkMarker`. This is because it is a sensitivity-neutral utility function that does not imply any assertion on whether the leak is acceptable. Otherwise, if line 12 is changed to Listing 4, `parse` no longer returns a security-sensitive value, which is incorrect behaviour.

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)));
```

3.2 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 at `build/reports/enclavlow/index.html` relative to the project on which task is invoked. The report contains the following elements:

Method summary Each method defined in the downstream project (i.e. excluding libraries and Java stdlib) is displayed with sensitive data that have passed through its parameters or return path. If multiple invokations lead to different data flows, the union of such data flows is displayed.

Redundant protection Methods detected to be run inside enclaves but never involved with any sensitive data are highlighted in an index called "Redundant protection". For each highlighted method, the developer should mark it as `@JOCall` to run out of enclaves or move its `@JECall` annotation to appropriate method calls, or adjust the `sourceMarker` / `sinkMarker` wrappers.

Data leaks Methods which result 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 "Data leaks". For each highlighted method, the developer should move it into enclave boundaries, or adjust the `sourceMarker` / `sinkMarker` wrappers.

3.3 Threat model

The adversary of concern has 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, the BIOS and hardware such as the CPU and the RAM, with the exception of the SGX execution part of the CPU. Note that untrusted hardware cannot execute the enclave code in the presence of cryptographically provable attestation performed with Uranus, [11], so privileged access to the CPU does not imply privileged access to the SGX module.

Since all applications of interest are run on Uranus, it is not meaningful to analyze threat models more capable than that as assessed by Uranus. In particular, side channels such as timing attacks are not to be assessed. *enclavlow* only studies attacks through at the data layer, where the adversary has read and write access to arbitrary data and instruction memory beyond SGX enclaves.

4 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.

4.1 Design

Since the adversary 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 propagte 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.

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

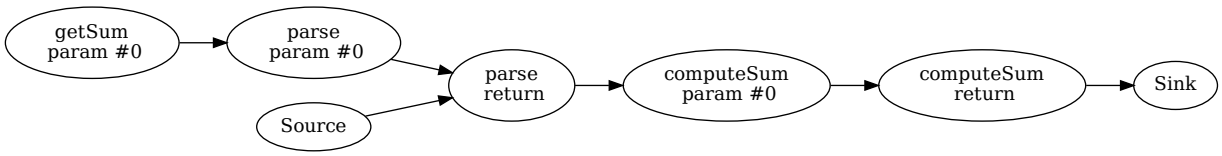
Note that this definition ("our definition") differs slightly from the usual definition of flow graphs (especially in DTA), where an edge (x, y) represents the flow of information from x to y [15]. In the usual definition, only adversary read access to y is concerned, while in our definition, the adversary possesses write access to x.

4.1.1 Contract flow graph (CFG)

enclavlow adopts an approach where a CFG is constructed for each method analyzed. The CFG contains the following nodes:

- "Static": Represents data located in static class fields
- "This": Represents the object on which a method was invoked
- "Param x": Each parameter is represented by a node

Figure 1: Example CFG for Listing 3



- "Return": Represents data flow through the return path
- "Throw": Represents data flow through the return path
- "Source": Represents variables in the method explicitly wrapped in `sourceMarker`
- "Sink": Represents locals in the method explicitly declared as `sinkMarker`
- "Control": This is a special node representing how many times the function is called.
- "This", "Param y", "Return", "Throw" and "Control" from each function called from the current node

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 aggregate flow graph (AFG), joined using the function call nodes.

For the case of OOP polymorphism, call graph analysis is performed to identify the exact subclasses that could be passed. In case multiple subclasses are possible, their contract graphs are merged by taking the union of all flow edges.

See Figure 1 for an example of AFG, with unconnected nodes omitted.

4.1.2 Local flow graph (LFG)

To construct the CFG, a local graph is constructed. The analysis follows along the control flow of the program, performing the *consume*, *branch* and *merge* operations.

The LFG extends the CFG with the following additions:

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

The *consume* operation consumes statements in form of 3AC [10]. Every step adds or removes some flow edges, as described exhaustively in Table 1. The relationship between the graph and the "lvalue"/"rvalue" terminology in Table 1 are explained in Table 2.

The *branch* operation performs a deep clone of the LFG and continues following each branch with its clone.

The *merge* operation pops the uppermost control flow node from the graph, and takes the union of all flows from each branched graph.

Table 1: 3AC instructions affecting LFG

Instruction type	Effects on LFG
Assignment	"Control" flows to lvalue nodes of destination. Erases current connections to lvalue nodes of destination. rvalue nodes of source flow to lvalue nodes of destination. lvalue nodes of source flow to rvalue nodes of destination.
Return	"Control" flows to "Return" node. rvalue nodes of returned value flow to "Return" node.
Throw	"Control" flows to "Throw" node. rvalue nodes of thrown value leaks to "Throw" node.
Conditional (If/Switch)	A new "Control" node is pushed to the control stack. Previous "Control" flows to the new "Control". rvalue nodes of predicate leaks to the new "Control".
Method call	Same effect as assigning call result to a sink variable.

Table 2: lvalue and rvalue nodes for expressions

Expression type	lvalue nodes	rvalue nodes
Binary operations	Unreachable	Union of rvalues from operands
Array literal `new int[a]` or `new int[]{a}`	Unreachable	Union of rvalues from count or literal elements
Array access `a[b]`	lvalues of `a`	rvalues of `a` and `b`
Instance field access `a.b`	lvalues of `a`	rvalues of `a`
Static field access `Class.field`	"Static"	none
Parameter	the parameter node	the parameter node
Local variable	its own dedicated node	its own dedicated node
`this`	"This"	"This"
Class cast and instanceof	lvalues of the underlying value	rvalues of the underlying value
Method/constructor call	"Return" of the called method	"Return" of the called method

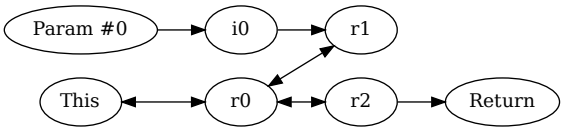
Note that the control flow stack is always pushed from a conditional instruction before splitting into branches, which is important for attacks that count the number of times a method was called, hence inferring the sensitive value that determined the branch halting problem.

In traditional information flow analysis, this naive approach described in the table appears to result in high false positive rate as it does not separate the internal structure used by instance fields and arrays. For example, consider Listing 5 (Jimple code in Listing 6). At return point, the LFG becomes as shown in Figure 2. Intuitively, this appears incorrect; param #0 does not flow to `return` since it is just used for `this.bar` but not `this.qux`. Nevertheless, in the threat model where the adversary has access to modify any memory allocated outside the enclave, assignment may not always work as intended; if the `TraditionalFalsePositive` context was allocated outside the enclave, `this.bar` might be modified by the adversary to `this.qux` (even if they belong to different classes), hence leaking into the return value. Recall the definition of an edge in the LFG used in this project, where $a \to b$ implies b must to be protected if a is protected.

4.2 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 he behaviour analyzed is as explicit as possible, all test cases are written in Java.

Figure 2: LFG of `TraditionalFalsePositive.foo(int)` in Listing 5



Listing 5: Traditional false positive

```
1  class TraditionalFalsePositive{
2      Bar bar;
3      Qux qux;
4
5      Qux foo(int y) {
6          this.bar.x = y;
7          return this.qux;
8      }
9
10     static class Bar {
11         int x;
12     }
13
14     static class Qux {
15         int x;
16     }
17 }
```

Listing 6: Traditional false positive (Jimple output)

```
1  class TraditionalFalsePositive extends java.lang.Object
2  {
3      TraditionalFalsePositive$Bar bar;
4      TraditionalFalsePositive$Qux qux;
5
6      void <init>()
7      {
8          TraditionalFalsePositive r0;
9
10         r0 := @this: TraditionalFalsePositive;
11
12         specialinvoke r0.<java.lang.Object: void <init>()>();
13
14         return;
15     }
16
17     TraditionalFalsePositive$Qux foo(int)
18     {
19         TraditionalFalsePositive r0;
20         int i0;
21         TraditionalFalsePositive$Bar $r1;
22         TraditionalFalsePositive$Qux $r2;
23
24         r0 := @this: TraditionalFalsePositive;
25
26         i0 := @parameter0: int;
27
28         $r1 = r0.<TraditionalFalsePositive: TraditionalFalsePositive$Bar bar>;
29
30         $r1.<TraditionalFalsePositive$Bar: int x> = i0;
31
32         $r2 = r0.<TraditionalFalsePositive: TraditionalFalsePositive$Qux qux>;
33
34         return $r2;
35     }
36 }
```

As Uranus was only tested against Linux systems, this project does not intend to support other operating systems. Furthermore, due to classpath detection difficulties, only OpenJDK Version 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 those frameworks 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.

4.3 Flow analysis framework

Soot [10] was selected as the framework for conducting flow analysis. Although multiple existing flow analysis systems using Soot alrady exist, they are not designed against SGX enclave protection, but *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 takes Java bytecode files (`*.class`) as input and compiles them into a 3AC language called Jimple, which simplifies analysis work. The consume-branch-merge approach mentioned in the last subsection was inspired by the forward flow analysis interface in Jimple.

Other flow analysis frameworks were also considered, such as Joana [4] and JFlow [13]. Soot was chosen due to its distinctively thorough documentation and builtin support for call graph analysis.

4.4 Testing

This project uses JUnit 5 and `kotlin.test` framework to conduct both unit and integration tests. Test case classes are compiled together in the `:core:testClasses` task, which declares compile-time dependency on the API `Source` and `Sink` annotations.

5 Results and Discussion

5.1 Difficulties and Limitations

The principles of OOP imply that a method called may be swapped with 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 obejcts of unexpected subclass through the `@JEcall` boundary. Consider listing 7 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.

Despite optimizations and simplifications, it is still not possible to perform 100% accurate information flow analysis within efficient time complexity [14]. For example, this project merges conditional branches together by taking the union of flow graphs, resulting in easy false positive rates. Listing 8 enumerates a number of false positives incorrectly identified by *enclavlow*, which are not going to be fixed because of the unlikeness of use.

Listing 7: Example attack through OOP substitution

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

Listing 8: Known false positives

```
class KnownFalsePositives {
    /**
     * This is a false positive due to multiple return arms.
     *
     * The CFG contains an edge (secret -> Return).
     * But in fact, Return is always 1.
     * This is considered a minor use case not to be fixed,
     * because duplicated code in multiple return arms
     * is typically regarded as an antipattern anyway.
     *
     * However, this pattern is checked as a special case
     * if multiple arms return the same literal,
     * including string literals, int literals and boolean literals,
     * but not for class constant literals.
     */
    int foo(int x) {
        boolean secret = sourceMarker(1);
        if (secret) {
            return x;
        } else {
            return x;
        }
    }

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

It is also impossible to analyze further than the JNI level, since analyzing across JNI boundary implies the need to interact with a native analysis tool, which is entirely out of scope of this project. Since Uranus effectively denies system calls, the contract is always assumed to be $\{(p, \text{return}) : p \in \text{params}\}$.

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, JLS 9.6.1.2 explicitly stated that "an annotation on a local variable declaration is never retained in the binary representation" [6], so a method call based approach is used instead.

At the technical aspect, multiple technical challenges were encountered when using Soot. Since Soot was designed to be a command line tool, it does not support direct calling from other environments very well. In particular, the entrypoint of Soot API is the `soot.Main.main()` method, which is the standard CLI interface in Java. As a result, extra time is spent on clearing global states used by Soot. Furthermore, due to restrictions in the Soot framework, each tested Java method must be run in a separate JVM, resulting in poor testing performance.

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.

References

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

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

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

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

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

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

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

[8] Bell, J., and Kaiser, G. Phosphor: illuminating dynamic data flow in commodity jvms. OOPSLA '14, ACM, pp. 83–101.

[9] che Tsai, C., Son, J., Jain, B., McAvey, J., Popa, R. A., and Porter, D. E. Civet: An efficient java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 505–522.

[10] Einarsson, A., and Nielsen, J. D. A survivor's guide to java program analysis with soot. *BRICS, Department of Computer Science, University of Aarhus, Denmark 17* (2008).

[11] Jiang, X. J., Tzs, C., Li, O., Shen, T., and Zhao, S. 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).

[12] Lind, J., Priebe, C., Muthukumaran, D., O'Keeffe, D., Aublin, P.-L., Kelbert, F., Reiher, T., Goltzsche, D., Eyers, D., Kapitza, R., et al. Glamdring: Automatic application partitioning for intel {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)* (2017), pp. 285–298.

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

[14] Smith, G. Principles of secure information flow analysis. In *Malware Detection*, vol. 27 of *Advances in Information Security*. Springer US, Boston, MA, 2007, pp. 291–307.

[15] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. Panorama: capturing system-wide information flow for malware detection and analysis. CCS '07, ACM, pp. 116–127.