# The World’s Most Advanced Sandbox™

**Li Haoyi** **Tim Kaler** **Frank Li** **Ivan Sergeev**

## Abstract

To safely run untrusted Java code, the untrusted code should be sandboxed. Traditional sandboxes rely on non-portable operating system facilities to run the code in separate processes with restricted privileges, which may also incur a high performance overhead. In this report, we introduce TWMAS, a sandbox that does not make use of the operating system. Instead, it uses the Java SecurityManager and instruction-rewriting to safely sandbox untrusted code directly running within a single host Java Virtual Machine.

## 1 Introduction

The World’s Most Advanced Sandbox™ (TWMAS) is a portable sandbox designed to safely run untrusted Java bytecode. Unlike traditional sandboxes, many of which use non-portable operating system (OS) facilities to run the untrusted code in a separate process with restricted privileges, TWMAS does not make use of the OS. Rather, it provides a way for a host Java Virtual Machine (JVM) to execute untrusted code directly, using the Java SecurityManager to block access to dangerous capabilities (Filesystem, Network, System, etc.) and instruction-rewriting in order to bound the number of bytes of memory allocated and the number of instruction executed.

Since Java applications run on a virtual machine, there already exists ways to limit the capabilities of untrusted code. For example, the SecurityManager can be used to assign Unix-like permissions to an application, limit access to the network, and block dangerous system calls. However, it does not provide a way to limit the CPU usage or memory allocations performed by any section of untrusted code within a single JVM.

The traditional way to impose these limits is to run the untrusted code in a separate JVM. However, it can take on the order of hundreds of milliseconds to spawn a new JVM, which is a significant overhead when running a large number of small untrusted functions.

In order to run untrusted code within a single process one needs a means of limiting the amount of memory it can allocate to prevent it from crashing the process. Similarly, if running the untrusted code within the same thread, we need a way of ensuring it eventually returns control back to the host and does not loop forever.

### What

At it's heart, TWMAS provides a novel primitive that an application developer can use to run untrusted code:

```
        run(Runnable untrusted, long maxMemory, long maxInstructions)
```

This method is called by the *host* process, and will execute the untrusted Runnable with a limited number of instructions it can execute and a limited amount of memory it can allocate. If the untrusted Runnable overshoots these limits, a ResourceLimitException is thrown, e.g.

```
sandbox.runtime.ResourceLimitException: Too Much Instructions Use! Additional
3 + existing 49999 would exceed maximum 50000
```

This will return control back to the host, who can catch the exception and resume. This prevents the untrusted code from looping forever, or allocating memory until the process fails.

These limits are imposed by re-writing the instructions of the untrusted code, as well as any code that it touches (i.e. imported Java libraries), to insert resource accounting checks into the untrusted code, thereby forcing it to keep track of the number of instructions and the amount of memory it has used. The rewriting happens at class load-time through the Java Instrumentation API. All code which runs on our instrumented JVM is associated with an Account object, which keeps track of the maximum resources the code is allowed to consume.

In this way, we are able to safely load untrusted Java binaries into our application, and run them directly, while being confident that this untrusted code cannot do anything that could possible harm or compromise the Host JVM.

### Roadmap

In this report, we document the details of TWMAS. In Section 2, we show how to enforce resource access control using the Java SecurityManager. Section 3 describes limiting memory usage, while Section 4 describes limiting number of instructions executed. We discuss TWMAS compatibility considerations in Section 5, and future works in Section 6. We make concluding remarks in Section 7.

## 2 SecurityManager

Untrusted code should not have arbitrary access to system resources or system operations. For example, the untrusted code should not be able to arbitrarily read or write to files in the file system, connect to sockets, spawn threads, kill a process, or change the current permissions settings. Our sandbox must mediate access to these resources, only allow accesses specifically granted, and blocking all others.

### How

Java’s `SecurityManager` is a class that allows an application to implement an access control security policy. A policy file defines the policy, a whitelist of fine-grained permissions to access resources. When an application instantiates the `SecurityManager`, it passes as an argument the path to the policy file. From then on, the SecurityManager enforces access control as defined by the policy on all code executed within the same JVM. Note this means that there is a single `SecurityManager` instance per JVM. Additionally, since the policy is a whitelist, all resource access will be restricted unless specifically granted. Below is a simple example policy, allowing only reads from test.txt and socket connections only to `localhost` on port 7777.

simple_example.policy

```
grant {
      permission java.io.FilePermission "test.txt", "read";
      permission java.net.SocketPermission "localhost:7777", "connect";
};
```

In TWMAS, we instantiate the SecurityManager after compiling the untrusted code, but before executing it with `run()`. The primary reason for this order is to avoid adding extra permissions needed by the compiler to the policy. This means the policy can instead simply define the permissions only granted to the untrusted code. Note that deleting an instance of the SecurityManager or changing the permissions requires its own special permission. By not granting the untrusted code these permissions, the sandboxed code cannot modify its access control permissions and will be restricted to the defined policy. Thus, by using Java’s built-in SecurityManager, TWMAS is able to enforce fine-grained access control to system resources and operations.

## 3 Memory Counting

Memory accounting is the mechanism by which we limit how much memory a section of untrusted code can allocate. At a high level, TWMAS assigns an Account object to each section of untrusted code, which specifies limits on memory usage (and number of instructions, described later). We insert accounting checks before each bytecode instruction that allocates memory. The Account object tracks how much memory has been allocated and prevents the allocation if the limit has been reached.

### How

We consider four different bytecode instructions which can allocate memory. The `NEW` instruction which allocates memory for an object of a given class, the `NEWARRAY` instruction which allocates memory for a new one-dimensional array of a given type, the `MULTIANEWARRAY` instruction which allocates memory for a new multi-dimensional array, and the `INVOKEVIRTUAL`, `INVOKESTATIC` instructions which allocate memory as the result of the method calls `Object.clone()` and `Array.newInstance()`. Prior to each instruction that allocates memory we insert an INVOKESTATIC instruction that calls the Recorder.checkAllocation method. The checkAllocation method calculates the allocation size, and then withdraws bytes from the untrusted code's account. If there are insufficient bytes in the account, then a `ResourceLimitException` is thrown.

## Calculating Allocation Size

For the `NEW` instruction, the `checkAllocation` method is called with a String argument that stores the name of the allocated object's class. We estimate the amount of memory needed to allocate an object of a given class using a global map of classes to their field counts. Since some classes may not be loaded until after the `NEW` instruction, it is sometimes necessary to load the class for analysis during the `checkAllocation()` call to determine the field count. Once the class's field count is obtained, we estimate the size of the object to be 8 * numFields bytes.

## Example (Object Allocation)

```
// Has 7 fields, should withdraw 8 * 7 bytes.
String s = new String();
```

For the `NEWARRAY` and `MULTIANEWARRAY` instructions, the `checkAllocation()` method is called with the size of the primitive type being allocated and the dimensions of the array. During instrumentation, we map each primitive type (double, int, char, etc.) to its size in bytes. The size of object references may vary depending on the JVM, so we estimate their size to be 8 bytes.

## Example (Arrays)

```
// Should withdraw 4 * 1000 bytes.
int prepareForTrouble = new int[1000];

// Should withdraw 8 * 1000 * 1000 bytes.
double[][] makeItDouble = new double[1000][1000];
```

Finally, we consider the `INVOKESTATIC` and `INVOKEVIRTUAL` instructions which can allocate memory when invoking the `Array.newInstance` and `Object.clone` methods.

## Example (Special Methods)

```
// Should withdraw 10 * 8 bytes
java.lang.reflect.Array.newInstance(Object.class, 10);

// Should withdraw 100 * 8 bytes
long[] sheep = new long[100];

// Should withdraw sizeOf(sheep) bytes
long[] dolly = sheep.clone();
```

## Object registration

Many Java applications create a large number of short lived objects which are collected by the JVM's garbage collector. In order to support these applications, we deposit bytes back into the untrusted code's account when one of its objects is freed by the garbage collector. If we do not do this, code which allocates a large number of short-lived objects will still run into their memory limits, even though the objects they allocate have all actually been garbage-collected and the memory returned to the system.

The garbage collector runs in its own thread and collects objects from all untrusted code running in the JVM, and will not necessarily know which application allocated a given object. For this reason, we add a registerAllocation call after each allocation to associate the newly allocated object with an account. The `registerAllocation` method creates a `WeakReference` to the object and then registers that reference with a thread local `ReferenceQueue`. We maintain a mapping of `WeakReferences` to the allocation size in bytes of the referenced object. To determine this size, we maintain a thread local stack of allocation sizes which has a size pushed upon a call to `checkAllocation` and popped upon a call to `registerAllocation`.

Since allocations and registrations should be properly nested, this allows us to obtain the correct allocation size for a registered object without recalculation. When the garbage collector frees the object, the `WeakReference` will be inserted into the `ReferenceQueue`. When withdrawing bytes from an account that is approaching its allocation limit, we process the elements inserted into the `ReferenceQueue` and deposit bytes back into the account. To help the application avoid running out of memory, we also add a call to `System.gc()` whenever an account has used up 3/4ths of its available memory. This logic is contained in runtime/AllocatableResource.java.

### Example (GC - Pass)

```
// This allocates a large number of short lived LinkedList objects.
// Since only one LinkedList object is reachable at a time, the
// short-lived objects will be garbage collected and the
// untrusted code will not run out of memory.
LinkedList<Object> list = new LinkedList<Object>();
for (int i = 0; i < 1000000; i++) {
  list = new LinkedList<Object>();
}
```

### Example (GC - Fail)

```
// This allocates a chain of LinkedList objects. Since each object
// remains reachable, the garbage collector will not free them,
// causing the untrusted code to run out of memory.
LinkedList<Object> list = new LinkedList<Object>();
for (int i = 0; i < 1000000; i++) {
  LinkedList<Object> prev = list;
  list = new LinkedList<Object>();
  list.add(prev);
}
```

## 4 Instruction Counting

The instruction counting instrumentation in TWMAS prevents untrusted code from consuming unbounded compute time, like an infinite loop or more sophisticated CPU-intensive code would, and allows for finer granularity of control over the allowed runtime of untrusted code. Other applications of the instruction counting instrumentation include systems for monetizing or sharing CPU time, and for benchmarking the performance of algorithms.

In order to do this, we pepper the instrumented code with `checkInstructionCount()` method calls. The method call to `checkInstructionCount()` contains a single argument: the number of instructions to be executed in the next block of untrusted code. The `checkInstructionCount()` method checks if this count exceeds the allotted account for the untrusted code. If the limit has been reached, an exception will be thrown which will abort execution of the untrusted code and return control to the host.

## How

We chose to place the instruction count checks before untrusted code basic blocks for correctness and performance reasons, at the price of some code complexity. This approach requires two passes of the target bytecode -- one for analysis and one for instrumentation -- whereas the alternative of placing checks after basic blocks can be implemented with a single pass of the target bytecode. However, placing instruction count checks before has the advantage of never executing code that will exceed the instruction count limit, which is correct in that the account will never be overrun. A maliciously engineered long block of instructions also will be caught ahead of time and not executed, leading to less wasted computational resources and better performance in those cases.

The instruction count instrumentation operates on a method-by-method basis, within each class loaded through our custom class loader. The first pass over the target bytecode identifies the boundaries of basic blocks, the instruction counts within them, and which labels are actual jump targets. A boundary can be the start of the method, a label (which can be branch, switch, or try-catch-finally targets), or a control flow change (such as a method invocation, a conditional/unconditional branch, switch statement, or a return). These boundaries mark the entry points in the target code, and are the locations where instruction count checks should be inserted to correctly count the actual instructions executed during runtime.

The second pass over the target bytecode interprets the same boundaries, but instead of recording instruction counts between them, inserts `checkInstructionCount()` calls with the recorded instruction count for the following basic block. The first call is placed at the start of the method, and requires the count for the following basic block for the argument, which is why this approach requires the first pass of analysis. The next call is placed at the next observed basic block demarcation, using the instruction count for the following basic block for the argument, and so on.

The third piece of information recorded during the first pass of analysis over the target bytecode is used as a performance optimization during the second pass instrumentation. Not all JVM bytecode labels are targets of conditional/unconditional branches, switch statements, or try-catch-finally blocks -- some are debugging line number annotations, and others may refer to variables -- so not all labels are necessarily entry points in the target code. The first pass builds a list of labels it observes to be actual jump targets, that is, labels that are explicitly encoded in a branch statement, switch statement, or a try-catch-finally block definition. When inserting a `checkInstructionCount()` call at an entry point boundary, the second pass uses this list to collapse consecutive basic blocks that are demarcated by a non-jump-target label, so that only a single call to `checkInstructionCount()` is inserted for every block of consecutively executed instructions. This reduces the overhead of our instrumentation, and improves performance of the target bytecode.

## Example

The examples below illustrate the information collected during the first pass analysis and the resulting bytecode after the second pass instrumentation.

```
========== Original Java Source ==========

public class EvilCode {
    public static void main() {
        int x;
        /* Evil infinite loop */
        while (true) {
            x = 1;
        }
    }
}

========== ANALYZING EvilCode9main()Vnull ==========

/// Boundary: Start of Method, Count: 0 ///
   L0
/// Boundary: Label, Count: 0 ///
    LINENUMBER 6 L0
    ICONST_1
    ISTORE 0
    GOTO L0                             JumpTargetList += [L0]
/// Boundary: Branch, Count: 3 ///
    MAXSTACK = 1
    MAXLOCALS = 1

JumpTargetList: [L0, L3]

========== INSTRUMENTING EvilCode9main()Vnull ==========

       L0
--> SIPUSH 3
--> INVOKESTATIC sandbox/runtime/Recorder.checkInstructionCount (I)V
        LINENUMBER 6 L0
      *  ICONST_1
      *  ISTORE 0
      *  GOTO L0
        MAXSTACK = 1
        MAXLOCALS = 1
```

## 5 Compatibility

Apart from a number of toy programs that unit-test the functionality of TWMAS (infinite loops, allocating enormous arrays, trying to write to files) we have successfully used TWMAS to run Mozilla Rhino, a JavaScript interpreter written for the JVM. Rhino is packaged by default with most Java installations, and represents a project of considerable size and complexity.

We have unit tests which successfully demonstrate the usage of TWMAS to limit the execution of malicious *JavaScript* code running in Rhino. Both a *JavaScript* infinite loop as well as *JavaScript* unbounded allocation in memory are both successfully blocked by TWMAS, while other less-resource-hungry scripts execute perfectly

## 6 Future Work

### Bugs

There are still a number of outstanding bugs that need to be worked out. Some of them are due to the large number of (often undocumented) special cases in the JVM, some are due to the difficulty in avoiding re-entrant behavior, some are due to the intricacies of the java ClassLoaders, some are due to incompatibilities between different versions of the JVM. In particular:

- The Rhino JavaScript interpreter seems to behave inconsistently on our different machines: It works perfectly on Windows, but on Ubuntu it occasionally fails to provide an interpreter when `getEngineByName(“JavaScript”)` is called, instead producing `null`.
- The `InfiniteCatch` unit test, which tests the ability for the instrumentation to break out of multiple enclosing `try{...}catch{...}` blocks, so far fails to do so. We believe that it is possible: every time the untrusted code jumps to a `catch{...}` block it should immediately throw an new exception and jump to a higher `catch{...}` block, and there can only be a finite number of these so it will have to return control to the Host.
- There is no accounting for native calls. For example, `System.arraycopy()` could perform a large amount of work, but currently counts as one instruction.

### Additional Features

- Another approach to resource access control would be to stub out existing system libraries used to access sensitive resources. This would give the sandbox complete control over when and how system operations occur. For example, this approach would allow us to create a virtual file system by modifying all file paths to point to a path within the virtual root. However, this approach does require stubbing out all possibly sensitive libraries with our own “safe” versions. We did experiment with this approach, creating safe versions for several system libraries accessing files. However, covering all possible libraries is an extensive task we considered too large a scope for this project, and we leave it for future work. Instead we primarily rely on Java's `SecurityManager` class.

- Fully measure the performance implications of TWMAS. By instrumenting the bytecode and adding SecurityManager checks, the runtime of untrusted code should increase. However, the amount by which this increases is currently uncertain.
- Allow a greater range of Java applications to run within the sandbox. For example, applications JRuby and Jython make heavy usage of runtime bytecode generation which is not supported by TWMAS. In theory, it should be possible to intercept this bytecode and instrument it while still allowing these applications to run.

## 7 Conclusion

TWMAS thus provides a way for a developer to run untrusted code on the JVM without worrying about the code "running away" and failing to terminate or exhausting the memory pool. This is done in pure Java, allowing us to avoid using non-portable OS level functionality (processes, users, groups) or having to embed an interpreter into our Java application to run untrusted code.

As it stands, the code is a prototype. All the bugs mentioned above all still exist. Our blanket-ban on any sort of dangerous calls throughout the entire process makes it difficult to perform any useful work using the sandbox, as that requires somehow communicating with the outside world. Furthermore, we have not yet investigated ways to improve the performance of instrumented code. Currently, the inserted checks cause instrumented code to run noticeably slower.

Nonetheless, we believe that TWMAS sandbox shows a great deal of potential. It provides capabilities which are hitherto unheard of: a granularity of individual instructions when controlling the CPU-usage of an untrusted piece of code, as well as controlling the memory allocation down the individual bytes, all while running at "native" speed on the JVM.