Detour hooking functions in the Hotspot JVM

Vladislav Khovayko

Abstract

This paper will explore a potential solution to implementing the procedure of placing a detour hook on a Java function at runtime without the use of Java agents, pattern scanning, or JVMTI, allowing users to run arbitrary code before and after a Java function has been invoked, modify arguments and return values, and replace a Java function with a different one. This project aims to extend the scope of capabilities available to people who use JNI, and can also aid in the debugging, monitoring, and reverse engineering of Java applications.
The information in this paper is based on the HotSpot JVM version 8 51 for Windows 10, and may be subject to change for other versions of the JVM.

Definitions

JVM: The Java Virtual Machine, the program that runs a Java application.
JNI: Java Native Interface, an official library that allows for Java and native code to communicate during runtime. This library is included in most JVM distributions.
JNIEnv: A pointer provided by JNI that is unique for each thread, used to call the functions that JNI provides. Using the wrong JNIEnv may lead to a crash or deadlock.
oop: ordinary object pointer, a generic pointer to something in the JVM, used for objects, strings, and some JVM structures, the end user typically never works directly with these, but rather wrappers that contain them such as JNI's jobject, jmethodID, and jclass structures.
Method: An internal JVM structure that holds information about some Java function, this includes the invocation and adapter entries, the Method's flags, a pointer to the interpreter, and other variables.
JavaThread: One of the internal JVM structures used to define a thread, contains important variables like the current thread state and JNIEnv instance.
Method compilation: The JVM has multiple threads that compile Java functions to native code during runtime, the C1 compiler runs faster than C2, but C2 produces more optimized code, a function can be compiled multiple times and sometimes deoptimized back to an interpreted state at runtime.
Garbage Collection: A process for freeing and optimizing memory within the JVM.

Overview of detour hooking

The concept of detour hooking is well known among various hacking communities, it's when a process's code gets modified so that when a targeted function runs, the program's execution will jump to a different function, such as a user-defined one in an injected library, and then jump back to run the original function, this is often referred to as "head injection" since the hook runs at the function's "head". More complex methods of hooking exist that also allow for "tail injection", manipulation of arguments and return values, and the ability to prevent the target function from executing.

There exist many libraries such as Minhook that make the "head injection" process easy to perform with native functions, however no official or widely used library exists that lets you hook Java functions.

This paper will explore the internals of the JVM with the aim of creating a hooking library with all the capabilities mentioned above.

Exploring the JVM's code and structures

In order to place a hook that runs before a Java function is invoked, it is important to look through the JVM's source code and find places that run prior to a method's invocation, there are multiple JVM functions that could be hooked to achieve this, however hooking them directly would require a signature/pattern scan, which would change depending on the JVM version, meaning such an approach would have little to no portability.

Getting a structure's fields: Given a pointer to a structure, an offset can be used to obtain a field within it, the values of offsets may vary depending on the JVM version.

The JVM has an internal structure "Method" defined in src/hotspot/share/oops/method.hpp, every Java function has a unique Method instance that corresponds to it, this structure contains several very important fields that will allow for a hook to be placed without the use of signature scanning. An instance of a Method pointer can be obtained by dereferencing a jmethodID, with the jmethodID being obtained through standard usage of JNI.

Method states and fields

At the launch of the JVM, most if not all of the Java functions will be in the interpreted state, meaning their bytecode is executed by the interpreter, during runtime the C1 and C2 compiler threads may compile or inline a function. The JVM may also sometimes “deoptimize” a Method, meaning it gets converted from compiled back to interpreted.
A Method’s state is either interpreted or compiled, it may also become inlined when certain conditions are met, such as the method being very small and being called often.

Important Method fields:

- **_adapter:** A structure that contains the i2c and c2i adapter pointers used for modifying the registers and stack when an invocation needs to be bridged between the interpreter and a compiled Method.
- **_access_flags:** These flags specify some of the Method’s properties, modifying them can prevent a Method from getting compiled and inlined.
- **_i2i_entry:** A pointer specifying the interpreter that should be used to execute a Method, many methods share the same interpreter, it's possible for there to be multiple interpreters.
- **_from_compiled_entry:** When invoking a Method, if the caller is compiled, it will use this pointer. If the callee is also compiled then this points to the compiled code to execute, otherwise points the c2i adapter so the invocation can be sent to the interpreter.
- **_from_interpreted_entry:** When invoking a Method, if the caller is not compiled, it will use this pointer. If the callee is compiled then this points to the i2c adapter so the invocation can be sent to the compiled code, otherwise points to the interpreter.
- **_code:** If the Method is compiled, this points to the compiled code that should be executed. Has a value of NULL if the Method is not compiled.

The JavaThread structure located in src/hotspot/share/runtime/javaThread.hpp also contains important fields:

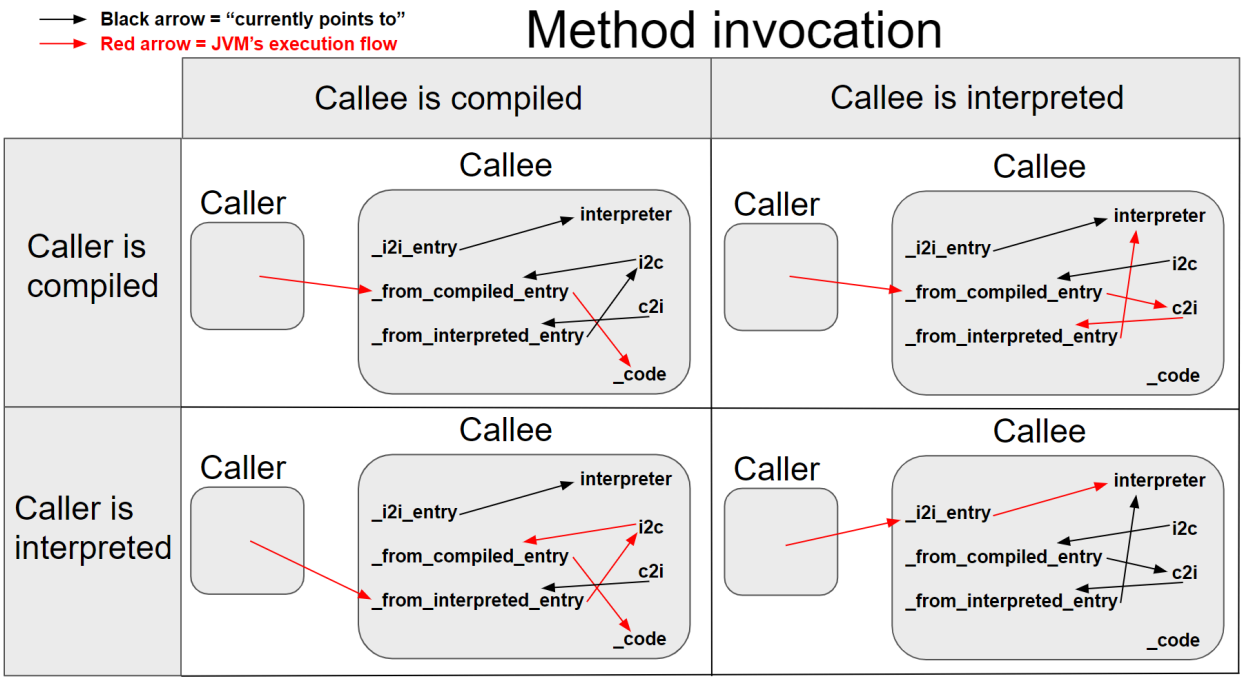
- **_thread_state:** This variable defines the current state that the JavaThread is in, having a wrong state may lead to a crash or deadlock of the JVM.
- **_jni_environment:** The JNIEnv instance that belongs to the current JavaThread is held in this field, by obtaining it the user will be able to perform JNI calls during the detour.

Getting offsets of fields

In order to use the various fields mentioned above, they must be accessed somehow, while it is known that they are located within the JVM structures, their offsets often change with the JVM version. Luckily the JVM exports gHotSpotVMStructs, which when iterated will list various fields within the JVM's structures and either their offsets or addresses if they are static. Not all fields get exported, so hard-coded offsets will be required at times.

Method invocation

The process of invocation depends on if the caller and callee is compiled or not, this picture shows the control flow that takes place during invocation:



_i2i_entry will always point to the interpreter.
_from_compiled_entry will either point to the **c2i** adapter or the compiled code of the Method.
_from_interpreted_entry will either point to the interpreter or the **i2c** adapter.
Whenever a method gets compiled, **_from_compiled_entry** will point to a new **_code** block.

Interpreter calling convention

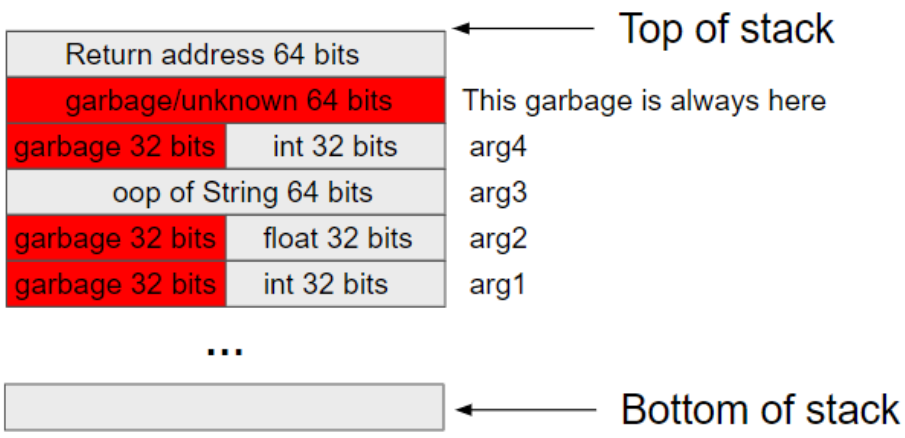
Due to multiple Methods sharing the same interpreter, the interpreter must know which Method instance it is currently working with, this is done by passing a Method pointer in the **rbx** register.
A pointer to the current JavaThread is held in the **r15** register.
The **r13** register holds the value that the stack must be restored to when the Method that was invoked returns, **r13** may hold a different value than **rsp** at the start of the invocation.
The JVM splits the stack into “stack frames”, with the current frame holding the return address to the caller and all arguments that are being passed to the Java function, arguments are stored in a backwards order.
When an argument is pushed on the stack, there will sometimes be garbage padding added on to the stack for alignment purposes.
Here are some examples of what the stack will look like at the start of a call to the interpreter:
Note: The “adjusted signature” is not an official or standard format, it's made to map the stack’s contents to a string, making it easier to understand and work with the stack.

Stack example 1:

static void funcA(int arg1, float arg2, String arg3, int arg4)

Java signature: (IFLjava/lang/String;I)V

Adjusted signature: (IFOI)V **In an adjusted signature every char takes up 64 bits**

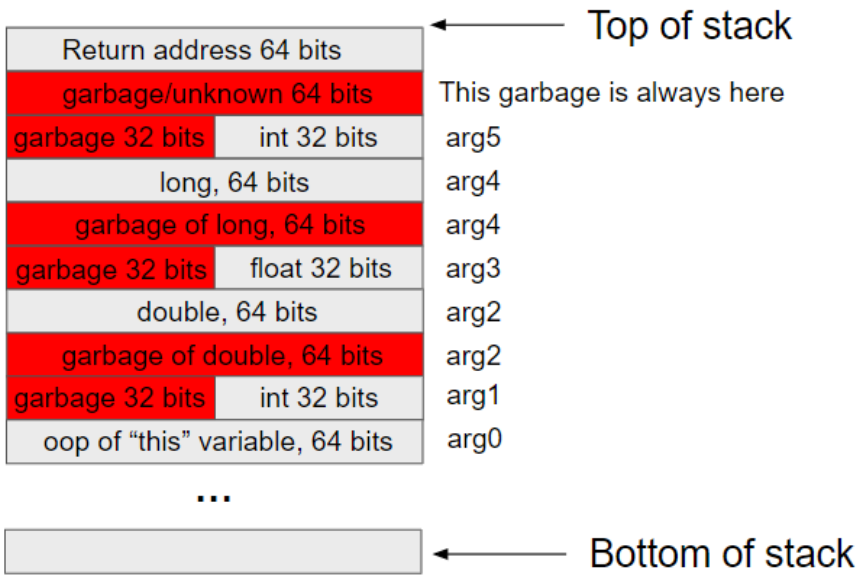


Stack example 2:

/*not static*/ void funcB(int arg1, double arg2, float arg3 long arg4, int arg5)

Java signature: (IDFJI)V

Adjusted signature: (OI_DF_JI)V **In an adjusted signature every char takes up 64 bits**



In some cases, it can be observed that some registers hold the arguments that are being passed to the Java function, this is due to them being placed there by the compiled calling convention before going through the c2i adapter, the interpreter will always use the stack rather than registers to get the arguments.

When an argument is pushed to the stack it will always take up a minimum of 64 bits, while doubles and longs will take up 128 bits.

Placing the detour hook

Now with a basic idea of the invocation process within the JVM, it should be possible to place a detour hook on a Java function of our choice, the first step is to obtain a Method pointer, this can be done by dereferencing a jmethodID instance, here is a code example:

```
jclass my_class = env->FindClass(“com/example/class”);
jmethodID my_method_id = env->GetMethodID(my_class, "foo", "()V");
unsigned char* method_ptr = *(unsigned char**)my_method_id;
```

Normally a Method can get compiled multiple times and even deoptimized back to an interpreted state at any moment, this makes placing a stable hook very challenging.
When placing the hook, the target Method may be interpreted, or it may have already been compiled or inlined, so the Method must first be modified to make it possible to hook it.
The JVM can be tricked into not compiling or inlining a Method by modifying the **_access_flags** variable, and adding the following flags: **JVM_ACC_NOT_C2_COMPILABLE**, **JVM_ACC_NOT_C1_COMPILABLE**, and **JVM_ACC_NOT_C2_OSR_COMPILABLE**.
Now the target Method will no longer compile, the next step is to set it to some state that can always be hooked, there is no guarantee the Method has been compiled yet, meaning its possible for the **_code** section to be NULL, so the only option is to revert the Method back into an interpreted state so that it can be hooked.
Reverting the Method into interpreted mode can be done as follows: set **_from_interpreted_entry** to point to the interpreter, set **_from_compiled_entry** to point to **c2i**, and set **_code** to NULL.
Now the target Method has been modified to be in interpreted mode and to never compile, and while it is now stable to hook, this does not mean the target Method’s structure will actually be used during its invocation, this is because if the target Method’s callers have already been compiled, they may have inlined the target Method, meaning they will no longer use the Method’s entries to call it.
In order to make sure the target Method is not inlined anywhere, the user must have a list of all of the Java functions that may call it, then for each Java function: set its Method to the interpreted state, but allow the JVM to compile them again if it wants to. This forced deoptimization will undo any possible inlining.

The end result is a stable Method that is always in the interpreted state and is not inlined anywhere, it's also known for a fact that all of its invocations will go through its entries and to the interpreter, meaning it is ready to be hooked.

Before placing a hook it's important to remember that the values of the registers must be correctly preserved, and since the interpreter has its own calling convention, this means that traditional hooking libraries such as Minhook will fail and lead to a crash.
In order to properly place the hook, a custom assembly function is needed that preserves all of the registers and can correctly return back to the JVM, once that is done we simply modify the **_i2i_entry** and **_from_interpreted_entry** pointers of the target Method to point to the assembly detour rather than the interpreter.
The following registers must be preserved: r8 to r15, rdi, rsi, rbp, rax, rbx, rcx, rdx, xmm0 to xmm7.
Register preservation can be done by pushing them to the stack and popping them back later, pushing the xmm registers may require the stack to be aligned first.

Now that registers have been preserved in the assembly detour, we must correctly return to the JVM, this can be done in two ways: return to the interpreter and have the Java function run, or simulate a return from the interpreter resulting in the function never running and the user now being able to return any value to the JVM.
Returning to the interpreter can be done by pushing the interpreter's address to the stack (this is what **_i2i_entry** originally pointed to), followed by a ret instruction.
Returning to the JVM can be done by popping the return address from the stack, setting **rsp** to **r13**, and jumping to the return address. Values are returned using the **rax** or **xmm0** register.

With a working assembly detour, its possible to add a call to a c++ function after pushing the registers, allowing c++ code to now run in the detour.
While being able to run c++ code in the detour is nice, there are a lot of limitations so far, in order to know the current Method that was invoked the **rbx** register's value must be checked.
In order to access the arguments for the Java function, the stack must be accessed as well.
Solution: In the c++ calling convention the first argument is put in the **rcx** register, also **rsp** essentially points to an array of values including all the registers, this means that if we copy **rsp**'s value to **rcx**, and define the c++ function to take a uint64_t pointer, we will now be able to

access and modify the stack, which contains the preserved registers and arguments given to the Java function.

Since the user may want to hook multiple functions at once, **rbx**'s value can be used to filter calls and make sure that they go into the correct user-specified detour function.

Here is pseudocode of the assembly and c++ implementation of the detour:

```
extern "C"{
    int _fastcall cpp_detour(uint64_t *stack){
        stack[RAX_POSITION] = saved_i2i_entry_value;
        for(hook h : hooks_list){
            if(h.method_ptr == stack[RBX_POSITION]){
                printf("Call came from %s\n", h.method_name);
                //can now call a user-specified function for this target Method
            }
        }
        return 0; //can be used to tell asm if to return to interpreter or JVM
    }
    void asm_detour(){
        asm(
            push registers r8 to r15, rdi, rsi, rbp, rax, rbx, rcx, rdx
            mov rcx, rsp
            push registers xmm0 to xmm7
            call cpp_detour
            pop registers back
            push rax //rax holds pointer to the interpreter now
            ret //return to the interpreter
        );
    }
}
```

With this, the detour is now implemented, all that's left is to implement the code that can get and modify the arguments that were passed to the Java function, as well as get a working JNIEnv to increase the detour's functionality.

Getting and modifying arguments

The layout of arguments in the stack was mentioned earlier, and now its possible to access them using a pointer to the stack. Arguments are stored in a backwards order (last argument is closest to the top of the stack), after the arguments are pushed they may be followed by other information, which was 128 bits for me (this might vary for other people), in addition to this there were 14 registers (64 bits each) pushed to the stack during the assembly detour.
This means that the last argument is located at stack_ptr[14+2], the adjusted signature mentioned earlier makes it easy to access all of the arguments in the stack.
Note that JNI's jobject is a wrapper for an oop, meaning oops from the stack must be converted to jobjects in order for JNI to use them, the Hotspot JVM provides the ability to convert an oop to a jobject using env->NewLocalRef(myOop);
Modifying an Object argument requires a jobject to be dereferenced to an oop for it to be put in the stack, the user must manually free the jobject later to prevent a memory leak.
It's important to remember that oops have a limited lifetime, and may be cleared by the Garbage Collector after the end of the function call, this means that saving the oops obtained from the arguments for later use can lead to undefined behavior since they may become invalid later, they must be upgraded to jobjects in order to do this.

Using JNI within the detour

As mentioned earlier, the r15 register contains a pointer to the current JavaThread, and the JavaThread structure contains a JNIEnv instance that we can retrieve and use, this JNIEnv instance should also be obtainable through a javaVM->GetEnv() call.
Unfortunately using JNI within the detour is unsafe due to the fact that it changes the JavaThread's **_thread_state** variable from **_thread_in_Java** to **_thread_in_native**, this leads to the Garbage Collector believing that it may free and relocate memory belonging to the current thread, normally the JVM prevents this with a complex system of locks, but the current detour implementation does not acquire them, which results in a crash if the Garbage Collector runs while the detour's code is executing.
The JVM does not provide any way to directly access its internal locks, the only way to acquire the required locks is to convince the JVM into placing them, the JVM does this when performing a call from Java code to native code, therefore the necessary locks can be acquired if the target function's Java code was swapped to new Java code that performed a native call.

All the required functionalities have already been implemented to do this swap, the first step is to use JNI to define a new Java class before placing any hooks, this class will contain two functions: a native bridge function that calls native code within the library, and a wrapper Java function that calls the native bridge function.

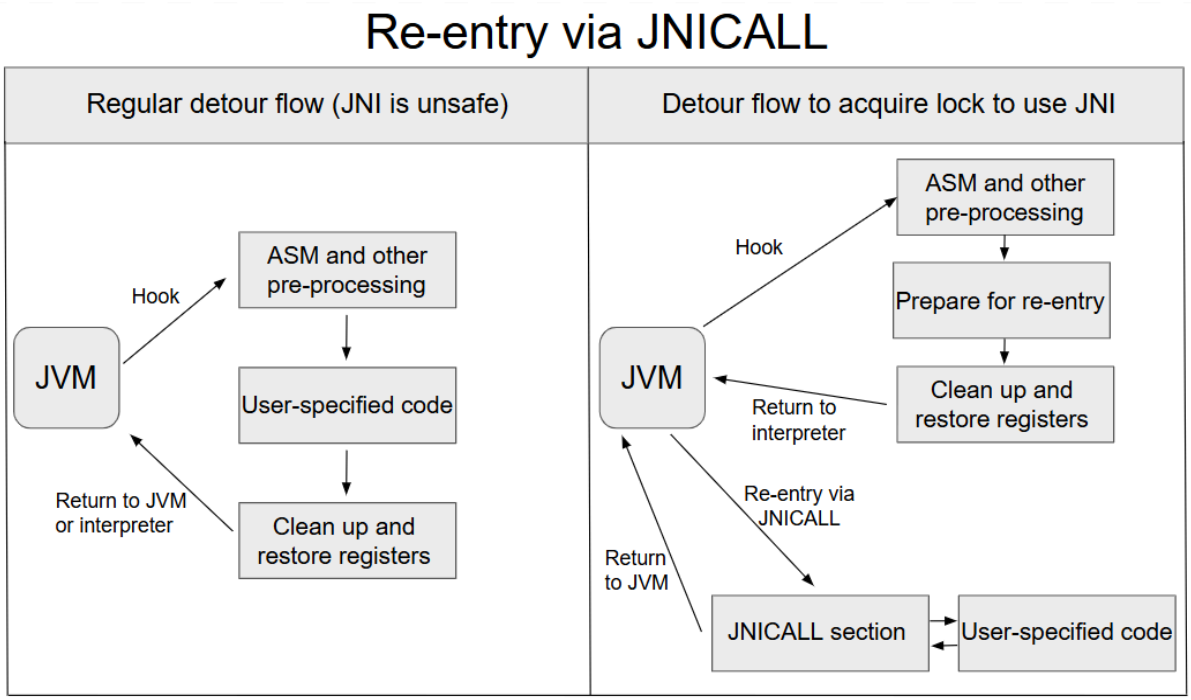
When the hooked Java function jumps to the detour the first time, the code will modify **rbx**'s value to now hold the Method pointer of the wrapper function, and perform a return to the interpreter, the interpreter will be tricked into executing the wrapper, which calls the native bridge function, places the necessary locks, and calls native code in the same library, this time JNI can safely be used.

Implementing the wrapper is a bit tricky in practice, the wrapper function must have the exact same signature as the hooked function, additionally if the hooked function is static, the wrapper must also be static, and vice versa.

Having a mismatch between the signatures will lead to the garbage collector cleaning up the arguments in the stack even with the necessary locks in place, however it is possible to edit the wrapper's signature at runtime for every call by digging into the Method's constant pool and modifying its entries, this will get around the restriction of having a matching signature.

Entries that need to be copied: **_max_locals**, **_size_of_parameters**, **_signature_index**.

This added complexity requires for there to be two wrappers for every return type, one static, and the other not. Each return type requires a unique native bridge function. Extra wrappers will be needed if there are a lot of different threads requiring to be bridged at the same time.

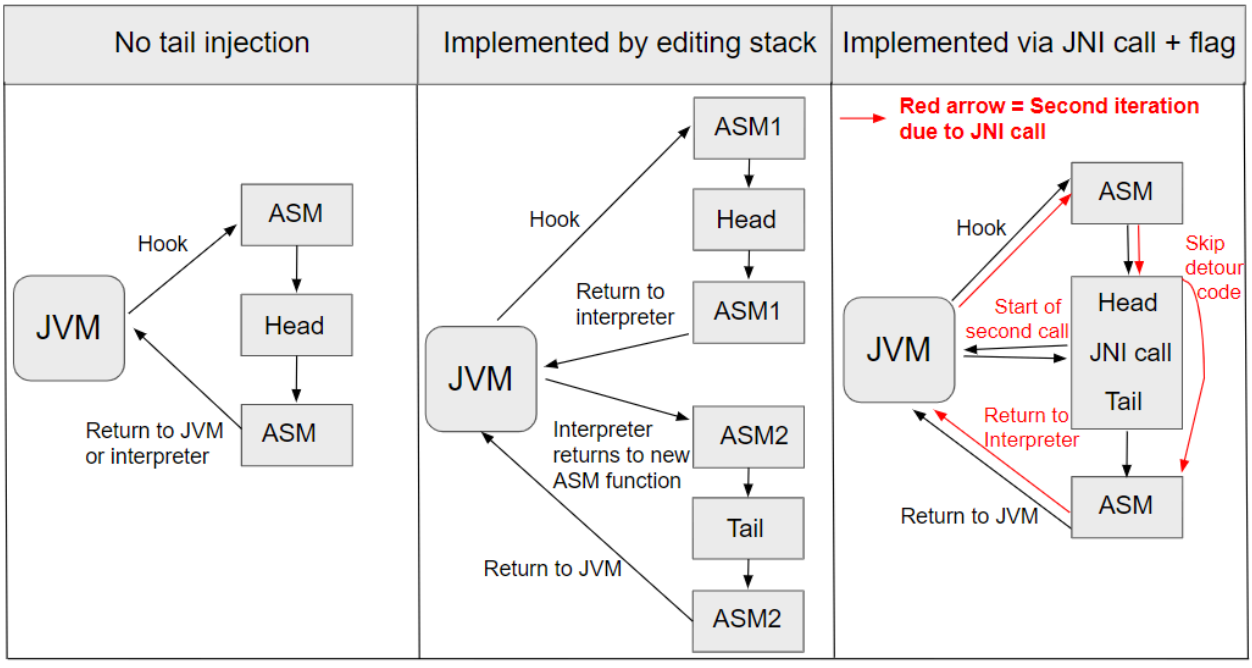


Implementing tail injection

Tail injection is the idea of running code after the target function has ran, but before it has returned to the process, there are two ways to implement this:

1. The first implementation involves either creating a new stack frame, or modifying the return address of the current stack frame, and returning to the interpreter, the interpreter runs the original Java function, and then using the stack frame's return address will return back to the user's code, letting them run code after the function's execution and modify the return value, the user should then return to the JVM.
2. The second way is easier but requires JNI: set a flag during the detour's first call, this flag indicates the current thread and Java function, the user uses JNI to call the original Java function, this call will be redirected back to the hook, check if the flag is set and if so: remove it and return to the interpreter, the interpreter will execute the original function and return to where JNI made the call from, letting the user run code after the function's call and modify the return value, the user should should then return to the JVM.

Visualization of tail injection:



Note: The method involving stack editing should be possible, but has not been implemented or tested.

Unhooking

The process of removing the hook is very simple: set **_i2i_entry** and **_from_interpreted_entry** to point to the interpreter that **_i2i_entry** was pointing to before the hook was placed, also restore the **_access_flags** field to its original value. Once all the detours have finished running, the library can be detached, if a class was defined (for JNI bridging) then it will remain loaded, there is no way to unload it, also if a bridge was used then this will prevent the library from being detached.

Future work

The implementation shown in this paper has the fatal flaw of working until it doesn't, there are thousands of different JVM distributions and each one will have its own set of unique differences, some of which will cause incompatibilities or perhaps make hooking completely impossible. There are likely better approaches to solve some of the challenges faced when creating this implementation that would be more flexible and less error-prone.

The current implementation modifies the constant pool of a user-defined Java class for every call that requires JNI usage in the detour, however a better approach would be to generate a class file during initialization that has all the necessary entry-point wrappers, this is a much less hacky approach that is still very flexible, additional classes can be generated during runtime if more functions need to be hooked that aren't known about ahead of time.

There is a small and usually unnoticeable performance impact caused by the target function being always set to an interpreted state, better performance might be achievable by being able to hook functions in their compiled state, although this seems to be either very difficult or impossible to do based on my current knowledge of the JVM.

The current implementation suffers from two known issues, the first being a false stack overflow, where the JVM believes a stack overflow has taken place, it can be triggered by performing a memory allocation within the detour such as: int* myint = new int; delete myint; , the second issue is JNI usage in daemon threads, they dont get properly protected even after going through the JNICALL process and may crash if the garbage collector runs during the detour’s execution.

Touch grass.

Conclusion

The lack of an official Java function hooking or callback system has caused a lot of inconvenience for many developers who use JNI, this paper shows that implementing something like this is quite possible, and can greatly increase the capabilities and options available to many developers.

This paper also has applications in the space of cybersecurity since it provides the ability to essentially hijack the behavior of any Java function, opening the possibilities for the creation of malware, software piracy, and a new powerful method for debugging Java applications.

Hopefully in the future the JVM developers will provide an official and standard interface to perform function hooking.

References, Related work, and Acknowledgements

[1] The unknowncheats.me community

https://www.unknowncheats.me/forum/minecraft/517132-dope-hooking-javas-interpreted-methods.html

[2] JVM Source code

https://github.com/openjdk/jdk17/blob/master/src/hotspot/share/oops/method.hpp

https://github.com/openjdk/jdk17/blob/master/src/hotspot/share/runtime/thread.hpp#L688

https://github.com/openjdk/jdk17/blob/master/src/hotspot/share/utilities/accessFlags.hpp

[3] LeFraudeur

https://github.com/Lefraudeur/

[4] Sebastien Spirit

https://systemfailu.re/

https://github.com/SystematicSkid

[5] ctsmax

https://github.com/ctsmax

[6] Daniel Strmecki

https://www.baeldung.com/jvm-tiered-compilation

[7] Iris Shoor

https://www.javacodegeeks.com/2014/01/mirror-mirror-using-reflection-to-look-inside-the-jvm-at-run-time.html