CHERI JNI: Sinking the Java Security Model into the C

David Chisnall[†] Brooks Davis[‡] Khilan Gudka[†] David Brazdil[⋆]
Alexandre Joannou[†] Jonathan Woodruff[†] A. Theodore Markettos[†] J. Edward Maste[◇]
Robert Norton[†] Stacey Son[□] Michael Roe[†] Simon W. Moore[†] Peter G. Neumann[‡]
Ben Laurie[▽] Robert N. M. Watson[†]
[†]University of Cambridge [‡]SRI International [▽]DeepMind Technologies Ltd.
[†]First.Last@cl.cam.ac.uk [‡]{brooks,neumann}@csl.sri.com [⋆]david.brazdil@gmail.com [◇]emaste@FreeBSD.org
[□]sson@me.com [▽]benl@google.com

Abstract

Java provides security and robustness by building a high-level security model atop the foundation of memory protection. Unfortunately, any native code linked into a Java program – including the million lines used to implement the standard library – is able to bypass both the memory protection and the higher-level policies. We present a hardware-assisted implementation of the Java native code interface, which extends the guarantees required for Java's security model to native code.

Our design supports safe direct access to buffers owned by the JVM, including hardware-enforced read-only access where appropriate. We also present Java language syntax to declaratively describe isolated compartments for native code.

We show that it is possible to preserve the memory safety and isolation requirements of the Java security model in C code, allowing native code to run in the same process as Java code with the same impact on security as running equivalent Java code. Our approach has a negligible impact on performance, compared with the existing unsafe native code interface. We demonstrate a prototype implementation running on the CHERI microprocessor synthesized in FPGA.

1. Introduction

Java was designed as a high-level language with strong type-safety guarantees and an integrated security model [11, 12]. The Java security model starts from a set of memory safety guarantees and supporting infrastructure such as bytecode verification, to ensure that running Java bytecode respects these guarantees. These guarantees are then used for building higher-level policies – for example, preventing any code that is not part of the `java.lang` package from directly invoking Java Virtual Machine (JVM) services. Other code must instead call methods on `java.lang.Runtime`, which will check the current `SecurityManager` and invoke the JVM services only if the calling code is permitted to do so.

These invariants are enforced while executing Java bytecode, but the JVM also includes standard interfaces for calling 'native' (typically C) code. The Java Native Interface (JNI) design document contains the following disclaimer [3]:

The JNI does not check for programming errors such as passing in NULL pointers or illegal argument types. Most C library functions do not guard against programming errors. For example, the printf() function usually causes a runtime error when it receives an invalid address, rather than returning an error code. Forcing C library functions to check for all possible error conditions would likely result in such checks being duplicated – once in the user code, and then again in the library. The programmer must not pass illegal pointers or arguments of the wrong type to JNI functions. Doing so could result in arbitrary consequences, including a corrupted system state or VM crash.

As such, any security and robustness guarantees provided by the JVM are lost when running software that contains, or depends on, native code. This is particularly unfortunate, as the reference implementation of the Java standard library includes around a million lines of C code exposed via the JNI [32].

A survey of the Google Play app store for Android found that 86% of the top 50 apps shipped native code [30]. A number of these (for example, Skype) use native code for a common core that is shared between multiple platforms – which presents an interesting target for attackers, as a vulnerability present in this code will be shared with iOS and Android (and possibly also desktop) versions of the same applications. A security study of the reference implementation of Java 1.6 [30] used static analysis to find 59 security flaws in native code, any one of which could bypass the Java security model.

ASPLOS '17, April 8–12, 2017, Xi'an, China..
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4465-4/17/04…$15.00.
http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/3037697.3037725

Multiple attempts have been made to sandbox native code (Section 4), using either separate processors or Software Fault Isolation (SFI) techniques [19]. These approaches either have high overhead or place significant restrictions on the functionality of the native code.

Given the importance of native code in the Java ecosystem, we propose than native code in an ideal JVM would:

- Run at the speed expected of native code.
- Be able to access any and all CPU features and do anything within its own memory space.
- Directly access buffers shared with the JVM without copying or interposition.
- Be constrained by the Java `SecurityManager` when accessing operating-system resources.
- Be unable to violate any of the memory safety guarantees expected by Java code.

We propose to use the hardware support for *memory capabilities* found in the CHERI ISA [41] (and present a prototype implementation) to support native code with these constraints and capabilities. CHERI's memory capability model provides support for fine-grained (object granularity) memory safety [8] and coarser-grained *compartmentalization* [36]. We use this to extend the Java security guarantees from Java code to native code running in the same process.

Our approach has several key advantages over prior techniques that have focused solely on sandboxing:

- Native code can safely include traditionally unsafe activities, such as run-time code generation and stack unwinders.
- We support efficient fine-grained robust sharing.
- We expose native-code sandboxing policy in Java, giving declarative control over sandbox lifetimes and state sharing.
- We support numerous sandboxes with diverse lifetimes.

We demonstrate that CHERI's primitives can be used to retrofit security to interfaces where it was explicitly not a design goal, and that the CHERI ISA supports compartmentalization models beyond those described in our prior work [36]. Figure 1 provides an overview of how sharing and communication are accomplished in our system.

2. Java Native Interface

The JNI is a mechanism to implement Java methods in C (or languages that implement the C ABI). When the native method is invoked, the JVM arranges a call frame containing C versions of the parameters, plus a pointer to a pointer to a structure that contains an array of function pointers for calling back into the JVM. Functions exposed via this interface include object creation, field and method access, and reflection.

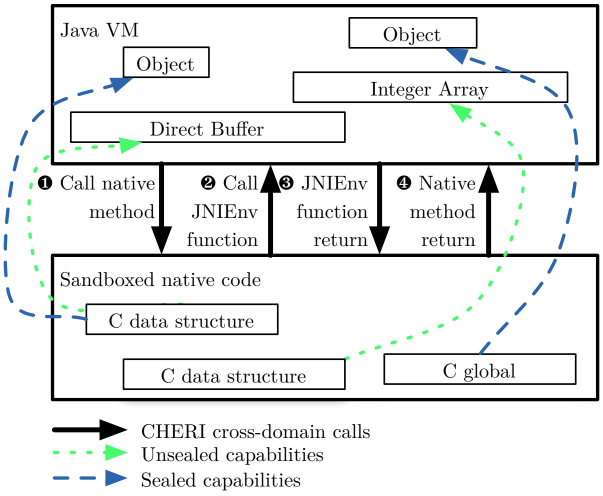

Figure 1. An overview of the CHERI JNI system.

Objects (including Java arrays) are passed to native code as an opaque `jobject` C-language type. The JVM specification makes no guarantees about object layout; if `jobject` is implemented as a pointer, native code must not use it to directly access object fields. When passed a reference to a Java array, the native code may request direct access to the underlying memory. The JNI is permitted to copy the data if the JVM does not support *memory pinning* (preventing specific allocations from being moved by the garbage collector).

The following example shows how the JNI provides a trivial native method that sums the elements of an array. In the Java code, the method must be declared with the `native` keyword:

```
class HasNative {
  native int sumArray(int[] arr, int len);
}
```

When this method is invoked, the JVM will search for a native function with a name that corresponds to an encoding of the class and method name (and types if it is overloaded), arrange the arguments in accordance with the calling convention defined by `JNICALL` (generally the normal C calling convention), and then jump to the function. This method might be implemented by the following C code:

```
JNIEXPORT jint JNICALL
    Java_HasNative_sumArray
  (JNIEnv *e, jobject o, jintArray arr,
      jint len) {
  jboolean isCopy;
  jint *a = (*e)->GetIntArrayElements(e,
      arr, &isCopy);
  jint sum = 0;
  for (jint i=0 ; i<len ; i++)
    sum += a[i];
  (*e)->ReleaseIntArrayElements(e, arr, a,
       0);
  return sum;
}
```

Even this simple example shows the fragility of conventional JNI implementations. If this method is invoked with a length greater than the length of the array, then a Java equivalent of this method would simply raise an exception. In contrast, this will read past the end of the array, potentially crashing the process. If the author of this C code had omitted the ReleaseIntArrayElements call, then the array object would be leaked. In slightly more complex C, a memory-safety error that involved writing through an out-of-bounds a-derived pointer would be likely to corrupt the Java heap. Errors of these forms and more have been found in native code included with the reference implementation of Java [30].

Java 1.4 introduced interfaces that expose access to the java.nio.ByteBuffer class, and guarantees that native code can directly access buffers [4]. This is a very important feature for Java scalability, because it is the underlying mechanism used with system calls for scalable non-blocking I/O – such as select, poll, epoll, and kqueue [4].

Java supports native code in Java applications for several reasons. In some cases, native code allows faster performance. In other cases, there are existing libraries in other languages; it is considered better practice to use them directly than to reimplement them. Cryptographic algorithms fit into both categories: they are often hand-tuned assembly or C code that is carefully designed to avoid timing vulnerabilities. Finally, some things are simply not possible in pure Java. For example, issuing system calls requires constructing a stack frame with a specific layout, and then issuing a special instruction – neither of which is permitted by pure Java code.

3. Threat model

Native code, such as many libraries used by the Oracle or Android Java stacks, is frequently used in place of Java code for performance reasons. This includes compression libraries, image and video decoders, and so on – as in C code written with performance as the overriding goal. Well-intentioned C code may contain memory safety vulnerabilities that enable an attacker to inject arbitrary code [31]. An attacker that is able to exploit vulnerabilities is able to run arbitrary code with the same permissions as the native code. In stock JNI implementation, the attacker has full control over the process. She can issue any system calls that the JVM is allowed to issue, or can stealthily corrupt Java objects.

In contrast, all Java code runs with a subset of this privilege controlled by the SecurityManager class. Before performing any action that should require privilege, the corresponding part of the Java class libraries will query the current SecurityManager, which will throw an exception if the current context does not have the required permission. This mechanism allows Java applications to run individual parts with reduced privilege. For example, when deleting a file from Java, the code that invokes the system's unlink system call will first call checkDelete on the current SecurityManager. In contrast, native code may simply invoke the unlink system call. This is also the case for Android apps, where setting a new SecurityManager will throw an exception, requiring the entire application to run with the same privileges.

Much native code requires significantly less privilege than would be useful to an attacker. For example, a native method decoding a PNG or JPEG image requires access to the input and output buffers and nothing else. The stock JNI does not provide a mechanism for enforcing the principle of least privilege. This includes both privileges that relate to access to parts of the process (may the code write to this bit of memory?) and external privileges (may the code write to files?).

4. Existing sandboxing approaches

Security was explicitly a non-goal for the JNI design. This has deterred attempts to retrofit security onto this model.

4.1 Process-based isolation

Sun Labs [9] moved native code into a separate process and communicated via RPC. This showed a desire for better security in native code, but had high overheads for production use. The cost of process creation is high, and the costs of copying data to and from the remote process were also high – in some cases an overhead of over 800%.

More recently, NativeGuard [30] provided tools for splitting Android applications into two separate processes. One process contains the Java code, and runs with the privileges that the application requires. The other contains the native code, and runs with a restricted set of privileges. These two communicate via an IPC channel. NativeGuard protects the Java program and the set of operating-system permissions that it holds from bugs in the native code (although the result may still be vulnerable to confused-deputy attacks). By supporting only a single sandbox process that has the same lifetime as the parent, the overheads of sandbox creation are amortized.

4.2 SFI techniques

The Robusta [28] and Arabica [29] projects are the state of the art in JNI sandboxing, and are very close to the best isolation possible with conventional hardware. Arabica provides a JVM-agnostic implementation of the techniques that the Robusta implemented as part of the JVM. Robusta uses Google Native Client (NaCl) [42] to provide a sandboxing substrate; it achieved low overhead (3-5% in the best cases, and 1,647% in the worst case invoking functions that executed very few instructions).

Robusta integrates Java security-manager checks into system call access for sandboxed code.

This work has several limitations inherited from NaCl sandboxing. NaCl relies on load-time verification of native code, and restricts the running binary to instruction sequences that are amenable to this verification. Some of the overhead arises from this, as the native code is prevented from using more complex addressing modes and idioms that would break the verification. To avoid an unfair comparison of MIPS and x86 implementations of a NaCl-like mechanism, we use a baseline for performance measurement that assumes the isolation guarantees of NaCl with no overhead other than the requirement to copy (rather than share) buffers. These limitations also prevent certain kinds of code from being run in the sandbox – for example, Just-in-Time (JIT) compilers and stack unwinders.

The most serious limitation is the lack of support for direct buffers. NaCl verifies that no memory accesses in the sandboxed code can reach outside of the sandbox. It cannot allow constrained accesses outside of the sandbox.

Finally, neither approach allows multiple sandboxes, nor any control over sandboxing policy from the Java code. A compromise in one native method therefore allows access to all data structures used by all native methods (including their stacks), and therefore can be used to subvert any native code. In addition, unsandboxed and sandboxed JNI methods use the same `native` keyword; it is very difficult to statically determine the sandboxing that has been applied.

5. CHERI

The CHERI Instruction-Set Architecture (ISA) [41] requires that all memory accesses are via *memory capabilities*, which extend 64-bit virtual addresses with metadata controlling their use: lower and upper bounds, permissions (read, write, execute, and so on), and a 1-bit tag that protects their provenance and integrity, coming to a total of 257 bits. Instructions to manipulate capabilities enforce a monotonic non-increase in rights (e.g., allowing removal of permissions or decreasing the range). Capabilities can be held in registers or memory; if store instructions overwrite a portion of a capability with non-capability data, the tag bit will be cleared, preventing further dereferencing.

All memory accesses are explicitly relative to a capability (using it as a pointer) or implicitly so (via the program counter capability for instruction fetches, or via the default data capability for loads and stores). This simple model allows us to define regions within a user address space where native code can run unrestricted, as long as it does not attempt to touch any memory outside of the bounded region.

CHERI also provides a *sealing* mechanism: a capability can be made immutable and non-dereferenceable (sealed) using another capability as a key. The sealed capability cannot be used to access memory until it has been unsealed with the same capability that was used to seal it. This is the foundation for CHERI's cross-domain call mechanism, which uses a pair of a code capability and a data capability, sealed with the same key, to describe a closure that can be invoked – without permitting the caller direct access to the code or data memory. This is sufficient to implement a NaCl-like model, without requiring static verification. Sealing allows pointers to be passed efficiently into native code and then be verified when passed back, but not dereferenced by the native code.

5.1 Memory-safe C

As described in our prior work [8], the CHERI C compiler supports two modes. In the pure-capability mode, all pointers (including function pointers) are represented as memory capabilities. In the hybrid mode, only specifically annotated pointers are represented as capabilities, and the rest are represented as 64-bit integers that are implicitly relative to one of the default capabilities. Code that lacks these annotations is binary-compatible with unmodified MIPS code (the native ISA).

Pure-capability bounds checking and pointer integrity provide some mitigation by themselves. Code-reuse attacks are difficult when the return address is protected by a tag bit. Attacks on the higher-level logic, or those that depend on temporal memory safety bugs, are still possible. Recompiling native code in this mode and using an existing JNI implementation would still allow the native code to damage the JVM.

We utilize the pure-capability mode in sandboxes, as it provides the greatest opportunity for exposing aspects of the Java security model to C. The protections afforded by CHERI will allow closer integration between high- and low-level languages, without sacrificing the guarantees afforded by the former. We can safely pass pointers to arrays on the Java heap directly to C code, yet prevent access to any of the rest of the Java heap. We can then use the tag bit to accurately find *all* pointers to the Java heap retained by a sandbox.

We use the hybrid mode for the JVM itself. Most JVMs perform a pointer-compression trick for object pointers on 64-bit systems, relying on the Java heap being within a contiguous range in the virtual address space. Java objects must typically be (at least) pointer aligned, meaning that the low three bits are implicitly zero – which allows a 32-bit integer to represent a 35-bit offset in a 16GB heap. We expect that a JVM designed for CHERI would use this trick, with object pointers within the JVM being offsets into a heap capability.

5.2 CHERI compartmentalization

CheriBSD, the CHERI adaptation of FreeBSD [20], includes libcheri – a library that manages compartments within a user address space [36]. Libcheri provides a class-based model for compartments. A libcheri class defines the layout of a compartment's address space, the set of functions that are exposed by it for cross-domain calls, and the set of functions that it expects the "system" object to expose for cross-domain calls. After a libcheri class has been loaded, libcheri objects may be instantiated from it. These object can then be used as targets for cross-domain calls.

Libcheri provides the `cheri_invoke` function and other bits of architecture-specific userspace code required for cross-domain calls. It also fills a role similar to the run-time linker with respect to cross-domain symbol resolution, creating vtables inside objects for dispatching incoming calls and assigning vtable indexes to method names for the caller.

The CHERI model provides a *trusted stack* of cross-domain calls. Every cross-domain call adds a record to the trusted stack, enforcing call-semantics. If a protection or segmentation violation signal is delivered inside a sandbox, then libcheri in combination with the kernel unwinds the trusted stack and allows the caller to recover.

6. Applying capabilities to the JNI

We use capabilities to provide an implementation of the JNI where native code can be placed into compartments in which it must respect all of the invariants of the Java memory and security model. Our prototype implementation uses JamVM, which is a simple bytecode interpreter; it works on big-endian 64-bit MIPS, the base architecture on which CHERI is an extension. Although this results in the Java code being slow, we do not consider this to be a problem for our evaluation, because we are investigating a standard interface between native and Java code, and believe that the lessons learned would apply to other JNI implementations.

6.1 Integration with the language model

The `native` keyword is adequate to describe native methods when there is no isolation, but lacks any description of policy for an implementation that provides compartmentalization. Java 1.5 introduced *annotations*, allowing arbitrary metadata to be associated with the methods (among other things). We declare a new `@Sandbox` annotation that describes a `native` method as being sandboxed. This annotation includes the libcheri class and the scope of the sandbox as attributes.

Libcheri provides a coarse-grained object model for CHERI compartmentalization. Libcheri classes are libraries that are loaded and instantiated to provide libcheri objects, with public functions (methods) that can be invoked using a cross-domain call. The `SandboxClass` attribute of the annotation tells the JVM which libcheri class to use.

The `scope` attribute is an enumerated type indicating a choice of method, object, or global scope for the sandbox. *Global scope* is the weakest and corresponds to the Robusta / Arabica model of a single sandbox for all native code using this libcheri class. *Object scope* means that a single sandbox exists for this pair (Java object and libcheri class); multiple native methods with the same libcheri class will share a sandbox when invoked on the same Java object, but methods invoked on different instances or with different libcheri classes will be isolated). Finally, *method scope* means that a sandbox is created for each invocation of the method, and destroyed (or, at least, reset) at the end of the invocation.

The global scope allows quick porting of existing JNI code. Both global and object scope methods allow state to be preserved between invocations (see Section 6.6).

Our earlier example, modified to use a method-scoped sandbox provided by the "sum" libcheri class would be as follows:

```
@Sandbox(scope=Global,SandboxClass="sum"
    )
native int sumArray(int[] arr, int len);
```

The C code would be unmodified. When executing, the C code from Page 2 would be invoked by a cross-domain call. It would then issue another cross-domain call back into the JVM via the `GetIntArrayElements` function, which would return a memory capability that grants access to the region of the Java heap containing the array elements.

If native code triggers a signal (e.g., a segmentation fault) that causes the trusted stack to unwind, then the JVM translates this into a Java exception and throws it. Similarly, if the native code calls the JNI with invalid parameters (e.g., requesting access to array elements in an object that is not an array), then the JVM unwinds two trusted stack frames (to return to the call from the JVM to native code) and raises an exception. In both cases, the JVM resets the sandbox state, as the sandbox is assumed to be in an undefined state.

We also extend the `java.lang.Runtime` object to include methods for explicit sandbox reset and revocation in global sandboxes. These can potentially leave sandboxes in an undefined state, and so are restricted by a security manager check.

The access to state between sandboxes depends on both the libcheri class and the sandbox scope. The Java access-control model permits objects to access the private data of other objects of the same class. With our model, you use a global sandbox with a libcheri class that provided only methods for a single Java class. You then have a single sandbox instance shared between all objects of the class, and therefore are able to share state via the C heap – but isolated from other Java components. Table 1 summarizes the settings that causes native code to share a sandbox. Two methods never share the same sandbox if they are provided by different libcheri classes.

6.2 Cross-domain calls

While implementing support for cross-domain calls, we encountered two limitations of the existing CHERI software stack. First, it did not provide a way of expressing a C-language function pointer that permitted cross-domain calls. To address this we adding a new `cheri_ccallback` calling convention to CHERI/Clang and a new built-in to construct a callback from a sandbox object and a function name. This is represented as a tuple of the sealed code capability, the data capability, and the method number for the target invocation.

Second, the strong security model from our prior compartmentalization work [36] implies mutual distrust for all cross-domain calls. In our model, there is asymmetric distrust: the JVM is assumed (perhaps erroneously) to be trustworthy, whereas the native code is not. For example, na-

	Same libcheri class			Different libcheri class		
	Global	Object	Method	Global	Object	Method
Same Java Object	✓	✓	×	×	×	×
Different Java Objects	✓	×	×	×	×	×

Table 1. Table of when different sandbox configurations share state.

tive code should be permitted to pass pointers to its stack to the JVM (although the converse should not be allowed). We addressed this by adding a second trampoline function and modifying the kernel's cross-domain call trap handler to skip the capability flow checks when invoked with a CHERI object with the high bit set in its type. We also modified libcheri to allocate types for system objects with this bit set, so that only designated system objects provided by trusted code are allowed to provide functions that bypass the checks.

6.3 JNI type safety

Accessing a field or calling a method on a Java object via the JNI is a two-step process. First, the native code requests a `jmethodID` or `jfieldID` from the JNI. Both of these types are defined to be pointers by the JNI specification. Second, the native code invokes another JNI function, passing back this pointer. In well-behaved code, the sequence from the native code looks something like this:

```
// Get the field ID for integer field x in
    class c
jfieldID f = (*env)->GetFieldID(env, c, "x
   ", "I");
// Set that field to 42 in object r
(*env)->SetIntField(env, r, f, 42);
```

In most current implementations, the JVM simply dereferences the pointer, trusting that the native code has passed a valid pointer of the correct type. Inside the JVM, the code that runs is typically like:

```
*(jint*)((char*)r + f->offset) = 42;
```

The Arabica [29] implementation (line 3335 of jinn/c2j_proxy.c) checks that the pointer is not null, but otherwise simply passes it back to the JVM's implementation. Malicious code can write to some arbitrary memory by producing a call like this:

```
struct field f = { ..., address - (long)r)
    };
// Set that field to 42 in object r
(*env)->SetIntField(env, r, &f, 42);
```

The JVM will write the value 42 to an address chosen by the attacker, bypassing the sandbox. This vulnerability is not intrinsic to SFI-based sandboxing, and could be addressed with a look-aside table indexed by `jfieldID`; however, this would introduce problems related to revocation, and impose an additional performance overhead.

We avoid this possibility by *sealing* each object pointer, method ID, and field ID with a different key when we pass it out to the native code. The hardware prevents the native code from modifying the sealed capability (because it does not have access to the key) and the JVM can cheaply check that the capability has the correct type when it is passed back. We also validate C strings passed from native code.

6.4 Array and buffer operations

The JVM provides a set of operations to request direct access to the underlying storage of a byte buffer or array. We support these by constructing a memory capability that grants access only to the buffer in the Java heap. The sandboxed code can then use this as any other pointer, but will receive a protection trap if it attempts out-of-bounds access.

Before passing these capabilities to native code, we explicitly remove the load-capability and store-capability permissions. This ensures that the native code may not store pointers in these buffers that it (or another sandbox) can later retrieve. We also remove the permission to store data from capabilities to read-only direct buffers.

There is one exception to this mechanism. Java arrays of Java objects may be accessed only via copy in our model. This is because the implementation of `jobject` (a memory capability) is different from the JVM's internal representation of a Java object reference. This would not be an issue for a JVM that used memory capabilities to represent Java references, although the trade-off space for performance versus robustness in doing this is beyond the scope of this paper.

6.5 System calls from sandboxed code

The CheriBSD kernel prevents sandboxed code from issuing system calls directly. Instead, each system call must be proxied by the surrounding environment. Libcheri previously supported only a cut-down libc and had manual support for a small number of system calls. We first separated the system-call layer from FreeBSD libc, allowing it to be replaced by a version that issued a CHERI cross-domain call with the same arguments. We then extended libcheri to provide implementations of these and hooks for code using libcheri to insert checks.

Interposition is a key policy primitive in capability systems [23], but poses challenges in the presence of concurrent resource sharing between mutually distrusting components [38]. To avoid race conditions, interposition policies must compensate for concurrent sharing, such as by copying pathname system-call arguments to non-shared memory before validating them and passing them on to the kernel. Similarly, where system-call arguments are references to process-scope resources (e.g., file-descriptor numbers), the safety and validity of those arguments must be enforced. In future work, we intend to explore mechanisms for allowing the kernel to implement this directly based on an application-provided policy.

Our sandboxes use a full libc and make system calls subject to policy from the JVM. The default behavior for all system calls is to return ENOSYS. We then selectively permitted them.

We began with the list of "safe" system calls from Capsicum [37]. These are not all safe in the context of sandboxing (for example, arbitrary use of mmap can be used to escape), but the majority are relatively benign. We protect these with checks for a RuntimePermission with a new "syscall") type.

We then protect the read and write system calls with checks for the existing "readFileDescriptor" and "writeFileDescriptor" RuntimePermission types. We also reuse both of these types for checking kqueue. Finally, we protect the open system call with the FilePermission check. In a production implementation, we would most likely provide hooks for more detailed checks.

For example, currently we completely prevent ioctl, because you must know both the type of the file descriptor and the operation to determine what it will do. When permitting direct access to hardware devices from sandboxed code, we would want Java programmers to provide a policy that restricts the sandbox to specific ioctl commands on specific file descriptors.

6.6 Revocation and garbage collection

Native code may hold references to Java objects, which must not be garbage collected as long as they are referenced. When attempting to extend the Java model into native code, we must extend the Java memory safety (including garbage collection) into native code as well. The constraints and implementation are slightly different, because the Java garbage collector does not have to support adversarial code generation: the Java interpreter and JIT cooperate with the GC.

Revocation and garbage collection are logical duals. Revocation ensures that references to an object do not remain after it should no longer be accessible; garbage collection ensures that objects do not remain after they are no longer accessible. We must provide both, because the semantics of the JNI APIs allow references that will extend the lifetime of objects as well as references that will become invalid (and must not remain – thus allowing use-after-free errors to corrupt the Java heap).

The JNI describes local references (which may not be retained past the current method invocation), and global references (which may be stored between invocations). As with the rest of the JNI design, it is the responsibility of the native programmer to ensure that these references do not persist too long in the native code. This is unacceptable for safe implementation. We support a safe version of this interface by tracking references that have been passed to native code and marked as no longer needed (i.e., global references that have been explicitly released or local references after the end of the method invocation). We also have to track references to pinned buffers to ensure that the underlying arrays or byte buffers are not deleted (or moved) while a sandbox has a reference to them.

Because CHERI allows the JVM to accurately locate all pointers to the Java heap in native code, an implementation using CHERI is not required to support pinning as long as the garbage collector rewrites all pointers in native code as well as those in Java. The continuously concurrent compacting collector [33] used in Azul Systems' high-performance Java implementation could be extended to allow fully concurrent revocation and collection in native code with our design.

A background JVM thread inspects all sandboxes that are not currently executing and scans the memory owned by the sandbox to find any of these that persist. By default, we provide 16MB of virtual address space per sandbox, and the operating system provides physical pages on demand. We use the mincore system call to check which pages have been touched. We thereby ignore any pages in the virtual address space that are not yet backed by physical pages.

All pointers in the native heap can be reliably identified by their (hardware-managed) tag bit, so a single branch can determine if a particular capability-sized data value is a valid pointer. We ignore anything that is not a capability or a reference to memory owned by the sandbox; we then have two cases to handle. If the object is sealed with a key identifying it as an object pointer, then we perform a lookup in a per-sandbox hash table to see if it is an object pointer that should have been removed. Alternatively, if it is unsealed and points to a range outside of the sandbox heap, then we search a red-black tree to find the first delegated array that starts at or before the start of this capability. We must do this slightly more complicated check because the native code is able to subset unsealed memory capabilities, and so may have derived capabilities to smaller ranges within an array.

If the capability that we have found matches an object capability or an array buffer to which the native code should no longer have access, then we revoke the capability by overwriting it with a NULL capability – which will cause a trap (and subsequent Java NullPointerException) if it is used.

Direct buffers present a challenge in this regard, as the JNI provides methods for directly accessing the memory owned by ByteBuffer objects, but does not provide a mechanism for informing the JVM that this access is no longer required. The native-code programmer is responsible for ensuring that the object persists for longer than the pointer. We pin these objects in memory until we have completed a scan of the sandbox and failed to find a capability to any of their buffer. We use the same mechanism to catch accidental resource leaks, where a native programmer has failed to call the JNI function that removes its reference to an object or array. This is implemented by a mark bit in the metadata for everything that we have delegated. After scanning the sandbox, we walk the hash table and tree, removing any unmarked entries, to implement a simplified mark-and-sweep collector.

The revocation policy has a similar set of constraints to a general garbage collection. We could run the revocation pass synchronously at the end of each method invocation (and did in our first prototype). This returns memory to the JVM as soon as it is no longer required by the native code, but comes with a significant penalty in the cost of a method invocation. In our later implementation, we record the number of times that a sandbox has been invoked and run the revocation in a background thread once this count has passed a threshold.

In a production-quality implementation, we would expect the revocation and garbage collection for native code to be tightly integrated with the Java garbage collector.

6.7 Compatibility versus performance

Our prototype implementation uses pure-capability code in the sandboxes. This gives the best security, but at the expense of some run-time overhead related to cache pressure from larger pointers. There are two other points on the spectrum that are worthy of consideration.

The first is to adopt the same design choices as Arabica, but use CHERI capabilities as the sandboxing mechanism – with unmodified MIPS binaries inside the sandbox. Unlike Arabica, this would allow stack unwinders and JIT compilers to run inside the sandbox. We did not pursue this approach because it would provide only small improvements over pure-software approaches. It would give us sandboxing, but would make various forms of delegation harder. We would not be able to provide direct buffer access, and object pointers would have to be indexes into a table of capabilities.

Table-based representations for object pointers make it difficult to spot accidental use-after-free errors. Consider the case where one part of the code informs the JNI that it no longer needs an object reference, but fails to zero it, and another causes the corresponding table index to be reused for another object. We would have no way of telling if a use of that object reference intended to use the first object or the second. With the pure-capability model, object references are distinct and can be accurately located on a native heap.

We also consider fine-grained memory safety within the C code to be advantageous, as it protects against a variety of programming errors. Sandbox setup initially grants the native code access to its entire heap space via the default data capability, so the fine granularity here is advisory. It is still possible to run code that is not memory safe with respect to its own stack and heap, but we gain protection against several vulnerability categories from well-intentioned yet buggy C code.

An alternative design would be to use the hybrid ABI and modify the JNI callbacks slightly, so that the direct buffer accesses would return `__capability`-qualified pointers. This would allow JNI code to use unmodified binary libraries, but would require source changes to the JNI portions. We rejected this design because we considered source compatibility to be of primary importance. It would be worth revisiting for situations where native code must use binary-only libraries, but where the JNI wrappers can be easily modified.

7. Evaluation

Our evaluation consists of both quantitative and qualitative portions. We first present an evaluation of the functionality and security benefits of our approach, then explore the costs.

7.1 Experimental setup

Our prototype uses JamVM, and the GNU classpath implementation of the Java standard library. The speed of Java code will be disproportionately lower than native code, when compared to desktop or server JVMs, which make use of advanced JIT compilation techniques (although with similar performance to JVMs intended for embedded IoT-like uses). To avoid biased results from this discrepancy, we restrict ourselves to comparing the performance of unsandboxed native code and sandboxed native code – eschewing comparisons with native-to-Java performance. This prevents us from demonstrating macro-benchmark results, where our overhead would appear very small as a result of the Java code taking a disproportionately long time to execute.

Our experimental platform is the CHERI processor synthesized for an Altera Stratix IV FPGA, a pipelined single-issue in-order 64-bit MIPS compatible core running at 100MHz, with 32KiB L1 caches, a 2-way set associative ICache and 4-way DCache, with a shared 256KiB 4-way L2 cache. The size of these caches match the L1 and L2 caches of Intel's Core architectures as well as the cache heirarchy of typical ARM Cortex A53 implementations. While absolute performance of this FPGA prototype is low compared to an ASIC, the architecture has been designed to yield measurement of representative relative overheads.

As with other JVMs, JamVM is multithreaded; thus, there is noise in our results from other threads preempting our benchmark thread, which we minimize through multiple benchmark runs. The error bars for the graphs are 99% confidence intervals from Welch's t-test.

Mechanism	JITs	Stack unwinders	Many sandboxes	Direct buffers
Process	✓	✓	×	×
Arabica	×	(✓)	×	×
CHERI	✓	✓	✓	✓

Table 2. Summary of the features of CHERI JNI sandboxing in comparison to state of the art implementations.

7.2 Functional evaluation

We have a set of functionality tests that check the various aspects of our implementation – for example, that the capability returned when native code requests access to a read-only `ByteBuffer` does not have store permissions. All tests pass.

We next demonstrate that we can implement those parts of the JNI that approaches focused purely on sandboxing have struggled with – as summarized in Table 2. In particular, we support all of the features required for zero-copy access to arrays and byte buffers, including read-only direct access.

In common with process-based sandboxing approaches, we support running arbitrary code inside the sandbox. This includes JIT compilers (for example, for JavaScript scripting in a Java application), which may generate arbitrary code. It also includes hand-written assembly code (e.g., cryptographic routines). In contrast, SFI-based approaches support running only code that passes the native code verifier.

The same applies to stack unwinders. Modern C++ implementations emit metadata describing the stack layout where values are spilled, which is then read by a library that walks the stack. NaCl (in Arabica) supports this specific use of a stack unwinder by providing a stack unwinder that runs outside of the sandbox (adding the cost of a domain transition to the cost of throwing an exception). This unwinder ensures that the target for a cleanup or catch handler is a valid code address. The unwind metadata format is defined by the sandboxing framework, and cannot be changed. In contrast, CHERI and process-based sandboxing will fault on invalid program-counter values.

Arabica listed support for multiple sandboxes as future work, but we have been unable to find evidence that this was completed. Supporting multiple sandboxes in process-based isolation runs into MMU and IPC scalability limits; it has been shown that CHERI scales significantly better [36].

No other approaches permit direct access to buffers, which the authors of the NIO specification [4] argued is essential for high-performance Java.

We are also able to use the CHERI sealing mechanism to provide efficient type checks on all pointers passed from the JVM to native code and back again. This avoids various type-confusion attacks that prior approaches could prevent only by incurring the overhead of a lookup-table mechanism.

7.3 Security evaluation

We sought to know how well our implementation would have protected the integrity of the JVM in the presence of past bugs in the use of native code with Java. To this end, we examined the vulnerabilities found in native code accompanying Java 1.6 by Tan and Croft's comprehensive study [32]. This study relates to an old version of the Java standard library, yet still provides some useful insight into the categories of vulnerability that can occur in mixing Java and native code. Their work found 126 bugs in JNI usage, of which 59 were security critical. We have examined which of these could have been prevented if the relevant methods had been sandboxed with our implementation.

11 of the 59 bugs were related to mishandling exceptions in native code. When Java code throws an exception, control is immediately transferred to the error handler. When native code throws a Java exception, the control flow transfer does not occur until the native code returns. We did not modify this behavior; however, the bugs in this category are no longer security critical. These all follow the pattern of checking that an operation will not overflow a buffer, throwing an exception if it will, and then forgetting to return and doing the unsafe operation anyway. In our implementation, these and five other buffer overflows would have triggered a forced unwind of the trusted stack and a `NullPointerException`, rather than a buffer overflow.

The next three security-critical bugs were race conditions in file accesses. These are all time-of-check-to-time-of-use errors. They would not have been prevented by our implementation, but we note that it would have been possible to produce similar errors in pure Java code – and so we do not regard them as bugs related to language interoperability.

The remaining 40 errors are related to insufficient error checking. Some of these are questionable, for example failing to check for `malloc` failure: On systems that perform memory overcommit such as Linux `malloc` will fail only if you run out of virtual address space, but may later tigger segmentation violations when accessed. In this case, we provide more robust checking, as the segmentation violation in native code would be caught and turned into a Java exception, as would the `NULL` pointer dereference in an implementation where `malloc` returned `NULL`.

The remaining 35 relate to missing `NULL` pointer checks or the kind of type-confusion attacks that described in Section 6.3. We would transform all of these into non-security-critical recoverable errors.

As such, *we would provide a defense against 56 of 59 security critical bugs* described by this study.

27 of the bugs that were not classed as security critical triggered memory leaks in Java code as a result of native code forgetting to release access to arrays. These may not be directly exploitable, but could lead to denial of service attacks. In our implementation, the garbage collector will detect that no references to the arrays remain and release them automatically.

This taxonomy did not investigate bugs in library code. For example, the Java version that they inspected used zlib 1.1.3, which contained a buffer overrun in the use of the `gzprintf` function (CVE-2003-0107). Inputs triggering this vulnerability would have caused a recoverable error in our implementation instead of memory corruption.

7.4 Performance measurement

We follow the example of the Sun Labs work [9] in using a simple microbenchmark of squaring a matrix to measure how the cost of the domain transition scales with respect to the amount of data and the resulting compute time. We wished also to explore the cost of memory copying, which is required by any attempt to sandbox native code without fine-grained memory safety. To explore the best possible case for such an implementation, we provided a version that has no sandboxing, but copies the input from the Java heap to the C heap before processing, produces results on the C heap, and then copies the results back to the Java heap. This shows the performance that would be achieved by sandboxing the native code using a hypothetical technology that had zero overhead for both execution time and domain-crossing, but no fine-grained memory safety.

The cost of using a global CHERI sandbox is dominated by a significantly higher cost of each domain crossing. This benchmark is implemented as a native method that takes three arrays (two for the input matrices, one for the output) and stores the cross-product of the two in the third. Each invocation therefore performs seven cross-domain calls: one for the JNI invocation, three each to get and release the array data.

The non-linear access patterns of this benchmark make it particularly susceptible to cache and TLB effects. Each run of the benchmark invoked the multiplication method 10 times and our results are from 200 runs for each size of matrix.

This benchmark demonstrates a high overhead (around 5×) for methods that do trivial amounts of work, which compares well with the published 7.2–16.5× overhead for similar workloads with Robusta and Arabica. There is also a high (although comparatively lower, at around 1.7×) overhead for simply copying the data in and out.

Figure 2 shows the extra overhead in cycle count for each run approach above the unsandboxed MIPS baseline. As the work done by the native method increases, the relative overheads decrease. By the time that we're performing a cross-product of a 50 × 50 matrix, the overhead for CHERI is $11.3 \pm 2.4\%$, compared to $9.9 \pm 2.2\%$ for the pure copying approach. In other words, *if another sandboxing mechanism had zero overhead but did not provide fine-grained memory safety, then it would not be faster than CHERI to a statistically significant degree* in this benchmark.

The CHERI implementation has a theoretical $\mathcal{O}(1)$ overhead from the cost of the more expensive domain crossings, whereas copying has $\mathcal{O}(n)$ cost in the number of elements in the matrix ($\mathcal{O}(n^2)$ in the size of one side).

In real hardware, the overheads are not quite the same. The cost of `memcpy` does not scale linearly with the size of the data being copied. It has a fixed cost for the function call and then discontinuities as the data size grows past cache and TLB sizes. Similarly, the CHERI case has a fixed overhead for each domain crossing, but has some secondary effects in perturbing the TLB caused by the sandboxed code using a stack and code in different pages to unsandboxed code. As the data size passes two pages, the TLB aliasing issues become more pronounced in both cases, and we see an increase in overhead for both. We also see a speedup in a small number of cases for the copying variant where the baseline suffers from cache aliasing between instruction fetch, the two input buffers, and the output buffers.

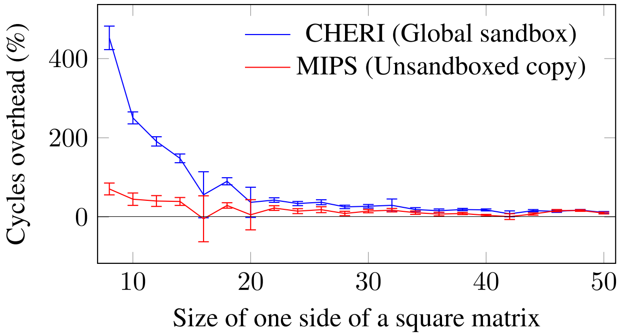

Figure 2. Matrix multiplication overhead relative to MIPS

We next explore the costs of real-world code for which security can be important. Zlib is a library with a less-than-stellar security track record; it is commonly used from Java for performance reasons. It is therefore a good candidate for using in a more secure way. The majority of the CVEs for zlib to date relate to the decompression path (for example, CVE-2005-2096, CVE-2003-0107, and CVE-2002-0059), and so we hypothesize that it is more dangerous. We therefore choose to place the compression path in a global (persistent) sandbox, but protect the decompression path by placing it in a method-scoped sandbox that is automatically reset between invocations.

Figure 3 shows the cycle overhead for compressing buffers of varying sizes with zlib, using the default compression settings. The native code requests direct access to two byte buffers (one for the uncompressed data, and one for the output) and their lengths, giving a total of four invocations of JNI functions from within the native method. The CHERI version uses a global (persistent) sandbox, the MIPS (copy) line shows a version that performs a redundant copy.

For larger data sizes, the sandboxed version runs around 20% *faster* than the unmodified MIPS binary, although it's worth noting that this speedup comes primarily from improved addressing modes in CHERI. Opcode space constraints required CHERI to have a single addressing mode for capability-relative loads and stores. This mode computes addresses from a capability register, an integer register, and a small immediate. In contrast, MIPS loads and stores take a single integer register and a (larger) immediate offset. In capability-rich code, CHERI's loads and stores can be more efficient.

This shows that relatively *minor changes to the ISA have significantly more impact on performance than our sandboxing model*. In the worst case (compressing only 32 bytes), we see around a 15% overhead (although there is significant variation between runs for this small size), which quickly drops off for larger sizes.

The version that performs a redundant copy is not always slower than the vanilla JNI. Close inspection of the performance-counter results indicates that this is likely to be caused by moving the buffer to a different address, which affects aliasing in the L2 cache. In the worst case, this shows just over 5% overhead, and in the best case a 1% speedup. This demonstrates that the overhead for our sandboxing approach is within the noise for small variations of memory layout and instruction-set design.

We contrast this with published results [29], which show overhead for Arabica with two JVMs, and for Robusta. The smallest data size that they benchmark is a 1KB buffer, which is already in the range where the CHERI version is faster than MIPS. The overhead of Arabica is around 20%, and Robusta just under 10%.

Figure 4 shows the overhead in decompressing the same data. *Every* method invocation in this benchmark will create and destroy a sandbox. This is an expensive operation, involving remapping memory for the sandbox heap, initializing any global capabilities, and so on. There is further room for optimization, but the current implementation provides a good worst case for comparison. Prior work has used persistent sandboxing; unlike the previous results, these cannot be directly compared with other approaches.

The costs of sandbox creation dwarf the costs of the work in the method for small quantities of data. Our smallest sample (decompressing 11 bytes and producing 32 bytes of uncompressed data) spends just under 3% of the time inside the sandbox, with the rest spent on creation and tear-down. This is obviously an extreme case: the overhead drops off rapidly as more time is spent inside the sandbox. It is also worth noting that, for small data, a pure Java implementation in a modern JVM would likely have adequate performance.

By 64KB of uncompressed data, the overhead is down to 15%; shortly above this we see that the overhead of a redundant copy exceeds the overhead of sandbox creation. This means that *any SFI or process-based approach would be slower above this size even if the cost of creating and invoking the sandboxes were zero*. This graph uses a log-log scale, and so the absolute overhead is hard to see. The important result to note on this graph is the highlighted crossover point.

Finally, we measured the *overhead on system calls*. A simple benchmark calls a trivial system call that returns a value without any locking, in a loop. We ran this benchmark in a global sandbox and with the conventional implementation. We also ran the sandboxed version with a trivial `SecurityManager` installed.

The absolute worst case for system call performance has no upper bound because the `SecurityManager` is allowed to run any arbitrary code. We saw an overhead of 422.8%±0.8% with no security manager and with a trivial one 455.0%±1.0% (with the usual caveat that Java execution in JamVM is slower than most Java implementations). This cost is fixed relative to the amount of work that the system call performs, and it would drop off quickly for system calls that do useful work.

7.5 Real-world applicability

We recorded two traces using ARM's Android simulator to begin to investigate how applicable these techniques would be in the real world. The first traces the Android boot, the second traces 30 seconds of a game of Angry Birds. Both record entry and exit to native code, and calls back to the JVM to access native arrays and direct buffers.

Figure 5 shows the distribution of array accesses (bytes shared per JNI invocation) in the Android boot trace, excluding around 100,000 calls that did not access any shared data. Some of these could be trivially elided if domain transitions were expensive (for example, `readBytes` with a `null` array or zero length is a no-op). Many others are not security-domain transitions; for example. `didPruneDalvikCache` is invoking a JVM service, not anything outside of the VM. The most common size for sharing in this trace is 40 bytes (which is the default size of a string object), but there is a long tail up to almost 400KB.

Many of the 8KB buffers are shared between native and Java code. These are in the region where the cost of copying or page flipping would often dominate, yet a CHERI domain transition would be cheap.

The Angry Birds trace was less informative. It appears that the game sets up sharing between native and Java code early on, and then simply passes notifications between the two.

Android already includes a special mode for the JNI (dubbed CheckJNI), which provides a subset of the guarantees of our implementation. This mode is enabled in debug builds, but is turned off for deployment because the overheads are too great. Running the Android Java test suite shows an overhead ranging from under 1% to almost 100% (double the run time of the normal JNI implementation), in pure Java workloads with the only JNI usage coming from standard library functions implemented in native code. The only times that we saw overheads this high with CHERI JNI were microbenchmarks of JNI invocations that were entirely dominated by domain transition overhead. We therefore believe that a CHERI design should approach the performance of the normal JNI, yet with a superset of the guarantees of CheckJNI.

7.6 Analysis of costs

To quantify our overheads, we recorded instruction traces for invoking an empty method via the sandboxed JNI. The results in Table 3 show that most of the cost is sandbox creation.

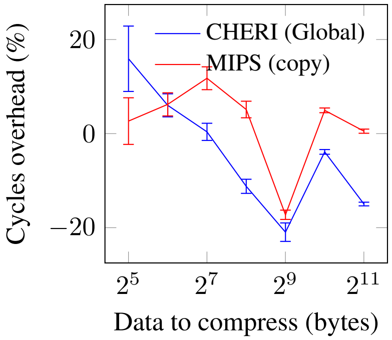

Figure 3. ZLib compression

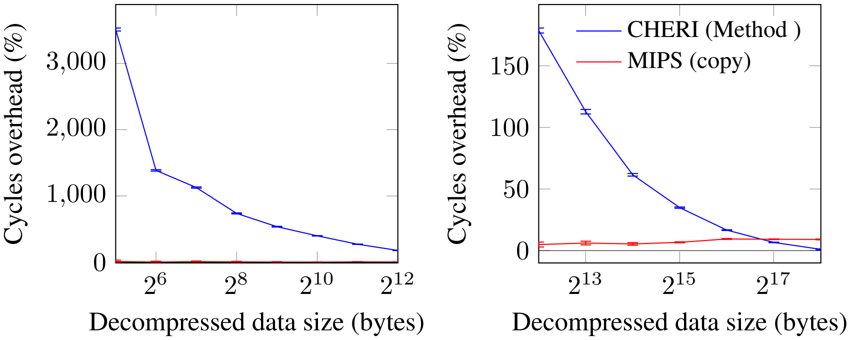

Figure 4. ZLib decompression

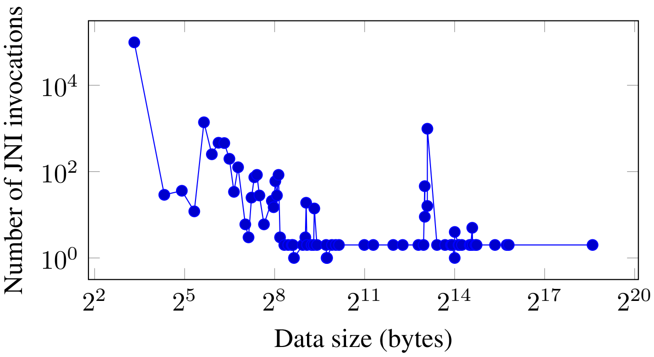

Figure 5. Sizes of array accesses in JNI invocations.

For persistent sandboxes, this is amortized across multiple invocations, but represents around 98.5% of the total instructions for the first invocation. The CHERI cross-domain call is a fairly minor part of our total overhead, at only around 12% of the total of the remainder. Most of the rest of the overhead comes from various bookkeeping activities. This code is not aggressively optimized, and uses off-the-shelf data structures (the FreeBSD red-black tree implementation and the UTHash hash table); thus, there is likely significant room for improvement in a production-quality implementation.

Stage	Instructions
Argument handling	160
Sandbox creation	216,204
GC bookkeeping	2,463
C argument frame setup	108
Cross-domain call	393
Java stack return	77
Sandbox Teardown	11,727
Total	219,405

Table 3. Instruction counts for invoking a sandbox.

8. Limitations of the JNI

Our experiences have led to a number of observations about the design of the JNI, which we hope will be of benefit to future language implementers. In spite of being explicitly designed without security in mind, we were pleasantly surprised by how easy it was to compartmentalize C code using this interface. There is no requirement that C code ever have direct pointers to the Java heap (with the exception of the direct buffer APIs, which may return "not implemented"). As a result, implementations where the native code is in a different process or in a NaCl or CHERI sandbox are possible.

The experiences are not all positive. For example, there is no type safety on method invocations from native code. On CHERI, we can use the tag bit, along with the sealing type, to differentiate object and non-object arguments from the caller. This avoids the most dangerous form of type confusion, where the Java code interprets a non-pointer value as a pointer.

Although useful for efficiency, the direct buffer access APIs make lifetimes implicit. This is problematic, as there is no explicit notification that native code has finished with the buffer; instead the native code is responsible for ensuring that the object persists for as long as it retains a reference. The `NewDirectByteBuffer` interface is particularly tricky, as the memory is allocated by the native code and must be freed by the native code (to avoid a memory leak), yet there is no mechanism for ensuring that the lifetime of the object exceeds the lifetime of the memory. We observe that `NewDirectBuffer` is rarely used. In the version of GNU Classpath that we are using, it is used twice in the same class (`java.net.VMNetworkInterface`) – a low-level class encapsulating information about a networking adapter. This class must perform operations that are not permitted to Java code, yet is not performance sensitive; restructuring the code to have the buffer provided by the Java code would be relatively easy.

9. Related work

We have discussed prior approaches to enforcing parts of the Java security model in Section 4. The obvious way to preserve the JVM's integrity guarantees in C code is to translate the C code to Java bytecode. This approach is taken by Jazillian (commercial, now defunct), Novosoft's c2j tool [5], and Tangible Software Solutions' C++ to Java Converter [6], although most have issues with some C constructs such as `union`s and `goto`. The various Ephedra publications [16–18] provide detailed coverage of this approach. A similar approach was taken by Microsoft in the various versions of C++ that target the CLR [7], providing both a subset (removing unsafe features) and superset (adding better integration with other .NET languages) of C++ that compiled to CIL. This approach provides a good way of migrating legacy code from C to Java; however, for new uses of native code, it must (for example) make system calls, or else generate executable bcode.

Other work has been conducted on providing memory safety to C. In hardware, this includes research projects such as HardBound [10] and commercial products such as Intel's Memory Protection Extensions [13]. Hardbound was followed by a software-only approach, SoftBound [24], and similar ideas were adopted by the Address Sanitizer [27].

Software compartmentalization has a long history based on a range of hardware- and language-based techniques. Karger proposed that fine-grained access control (via hardware capability systems) could mitigate malicious software [14]. Process-based privilege separation using Memory Management Units (MMUs) has been applied to FTPd [15], OpenSSH [25], Chromium [26], Apple iOS/Mac OS X [39], and in Capsicum [37] – although with substantial performance overheads and program complexity. More recently, hardware primitives such as Mondriaan [40], CHERI [36], and CODOMs [34] have extended conventional MMUs to improve compartmentalization performance and programmability.

Java sandboxing develops a mature and complex policy mechanism on top of language- and run-time-imposed memory safety [11], but leaves open the possibility of misbehaving native code. Language-based capability systems, such as Joe-E [21] and Caja [22], allow safe compartmentalization in managed languages such as Java – but likewise do not extend to native code. Software Fault Isolation (SFI) techniques [35], such as NaCl [42], transform code to impose coarse-grained sandboxing, but have difficulty expressing fine-grained memory sharing. As discussed in Section 4, process- and SFI-based native-code sandboxing for Java has encountered performance and expressiveness limitations.

10. Acknowledgements

We thank our colleagues Jonathan Anderson, Ross Anderson, Ruslan Bukin, Gregory Chadwick, Nirav Dave, Bob Laddaga, Alex Richardson, Colin Rothwell, Howie Shrobe, Munraj Vadera, Stu Wagner, Bjoern Zeeb for their feedback and assistance. We would also like to thank Şerban Constantinescu for providing traces of Android's JNI usage and measuring CheckJNI overhead. This work is part of the CTSRD and MRC2 projects sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 and FA8750-11-C-0249. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. We also acknowledge the EPSRC REMS Programme Grant [EP/K008528/1], the EPSRC Impact Acceleration Account [EP/K503757/1], Isaac Newton Trust, UK Higher Education Innovation Fund (HEIF), Thales E-Security, and Google, Inc.

11. Availability

We have released the CHERI hardware and software stacks, specifications, and manuals, as open source on the CHERI CPU web site [2]. Our experimental data are available at on the CHERI open data site [1].

12. Conclusion

With CHERI hardware assistance, it is possible to ensure that native code is forced to live within all security guarantees that Java expects – without modifying the native code.

We have demonstrated the feasibility of an implementation of the Java Native Interface that is safe, yet implements all of the features required to support the JNI. We have shown that extra copying – something that the Java NIO framework was intended to avoid – can have a greater performance impact than creating and destroying a sandbox on every method invocation. This highlights the benefits of providing fine-grained memory safety at an ISA level, allowing it to be used from C to maintain invariants that are expected from higher-level languages.

Even in our unoptimized prototype, overhead is negligible for persistent sandboxes, and for ephemeral sandboxes processing large amounts of data. We have shown that CHERI's fine- and coarse-grained memory protection are essential for efficient integration between high- and low-level languages.

We have extended the CHERI sandboxing mechanism to support callbacks and asymmetric distrust, demonstrating the flexibility of the CHERI instruction set in adapting to security models beyond those initially described. Our hardware-assisted implementation both supports more features and runs faster than a purely software implementation.

References

[1] CHERI open data web site. `https://www.cl.cam.ac.uk/research/security/ctsrd/data/`. Accessed: 2017-01-27.

[2] CHERI open-source web site. `http://www.cheri-cpu.org/`. Accessed: 2017-01-27.

[3] Java native interface specification. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html. Accessed: 2016-07-25.

[4] Jsr 51: New i/o apis for the java platform. https://jcp.org/en/jsr/detail?id=51. Accessed: 2016-07-25.

[5] Novosoft c2j. http://www.novosoft-us.com/solutions/product_c2j.shtml. Accessed: 2016-07-25.

[6] Tangible software solutions' c++ to java converter. http://www.tangiblesoftwaresolutions.com/Product_Details/CPlusPlus_to_Java_Converter_Details.html. Accessed: 2016-07-25.

[7] C++/CLI language specification. (ECMA-372), December 2005.

[8] David Chisnall, Colin Rothwell, Brooks Davis, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Peter G. Neumann, and Michael Roe. Beyond the PDP-11: Architectural support for a memory-safe c abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 117–130, New York, NY, USA, 2015. ACM.

[9] G. Czajkowski, L. Daynes, and M. Wolczko. Automated and portable native code isolation. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 298–307, Nov 2001.

[10] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. *SIGPLAN Not.*, 43(3):103–114, March 2008.

[11] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the Symposium on Internet Technologies and Systems*. USENIX, December 1997.

[12] Li Gong. Java security architecture revisited. *Commun. ACM*, 54(11):48–52, November 2011.

[13] Intel Plc. Introduction to Intel memory protection extensions. http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, July 2013.

[14] P.A. Karger. Limiting the damage potential of discretionary Trojan horses. In *Proceedings of the 1987 Symposium on Security and Privacy*. IEEE, April 1987.

[15] Douglas Kilpatrick. Privman: A Library for Partitioning Applications. In *Proceedings of 2003 USENIX Annual Technical Conference*, 2003.

[16] Johannes Martin. *Ephedra - A C to Java Migration Environment: Approaches, Case Studies and Tools for Migrating Legacy Systems from C and C++ to Java*. LAP Lambert Academic Publishing, Germany, 2009.

[17] Johannes Martin and Hausi A. Muller. Strategies for migration from c to java. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, CSMR '01, pages 200–, Washington, DC, USA, 2001. IEEE Computer Society.

[18] Johannes Martin and Hausi A. Müller. C to java migration experiences. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, CSMR '02, pages 143–153, Washington, DC, USA, 2002. IEEE Computer Society.

[19] Stephen Mccamant and Greg Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. Technical Report MIT-LCS-TR-988, May 2005.

[20] Marshal Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Pearson, 2014.

[21] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A Security-Oriented Subset of Java. In *NDSS 2010: Proceedings of the Network and Distributed System Security Symposium*, 2010.

[22] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized javascript, May 2008. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

[23] Mark Samuel Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006.

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[25] Neils Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*. USENIX, 2003.

[26] Charles Reis and Steven D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*. ACM, 2009.

[27] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012.

[28] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the native beast of the jvm. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 201–211, New York, NY, USA, 2010. ACM.

[29] Mengtao Sun and Gang Tan. *JVM-Portable Sandboxing of Java's Native Libraries*, pages 842–858. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[30] Mengtao Sun and Gang Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless &#38; Mobile Networks*, WiSec '14, pages 165–176, New York, NY, USA, 2014. ACM.

[31] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[32] Gang Tan and Jason Croft. An empirical security study of the native code in the jdk. In *Proceedings of the 17th Conference*

on Security Symposium, SS'08, pages 365–377, Berkeley, CA, USA, 2008. USENIX Association.

[33] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. *SIGPLAN Not.*, 46(11):79–88, June 2011.

[34] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting software with code-centric memory domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 469–480, Piscataway, NJ, USA, 2014. IEEE Press.

[35] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles*. ACM, 1993.

[36] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, May 2015.

[37] R.N.M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for Unix. In *Proceedings of the 19th USENIX Security Symposium*. USENIX, August 2010.

[38] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *WOOT '07: Proceedings of the first USENIX Workshop on Offensive Technologies*, pages 1–8, Berkeley, CA, USA, 2007. USENIX Association.

[39] Robert N. M. Watson. A decade of OS access-control extensibility. *Commun. ACM*, 56(2), February 2013.

[40] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[41] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: revisiting risc in an age of risk. In *ISCA '14: Proceeding of the 41st annual international symposium on Computer architecture*, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[42] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.