CHERI JNI:
Sinking the Java security model into the C

The Java abstract machine

Java VM
- Type safe
- Garbage collected
- Data hiding
- Security policy enforcement

Native Code
- Fast
- Full control over memory
- Basically full of evil

JNI
The Java concrete machine

Native Code

Java VM

Type safe
Garbage collected
Data hiding
Security policy enforcement

JNI

Fast
Full control over memory
Basically full of evil
JNI is insecure by design

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...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.
BRIEF CHERI PRIMER
Hardware memory safety

Compressed representation provides the same abstract model in 128 bits.
Sealing gives opaque pointers

Trust code

Untrusted code

Trusted code

Data Capability

Sealing Capability

Seal

Sealed Capability

Unseal

Other data

Unseal

Trusted code

Untrusted code

Trusted code

Data Capability

Trap!
CHERI AND THE JNI
CHERI JNI

Java VM

Type safe
Garbage collected
Data hiding
Security policy enforcement
Declarative policy for sandboxed native code.

JNI

Native Code

Fast
Full control over local memory
Memory-safe access to Java buffers
Controlled access to Java objects
Evil stays in the box

JNI

Native Code

JNI

Native Code

JNI

Native Code
Sandbox scopes

- Fast
- Global
- Object
- Method
- Secure
- Easy
Direct buffer access

JNIEEXPORT void JNICALL
Java_Foo_process
  (JNILong  *env, jobject this, jobject buf) {
    char *b = (*env)->GetDirectBufferAddress(env, buf);
    someNativeLibraryThing(buf);
  }
Direct buffer access

class Foo {
    @Sandbox(...)  
native long process(ByteBuffer buf)
}

JNIEXPORT void JNICALL
Java_Foo_process
(JNIEnv *env, jobject this, jobject buf) {
    char *b = (*env)->GetDirectBufferAddress(env, buf);
    someNativeLibraryThing(buf);
}

Bounds-checked access to JVM-owned buffer and nothing else.
No store permission if the ByteBuffer is read only.
Avoiding type confusion

Exploitable vulnerability in existing state of the art SFI-based technique.

JVM does:
\[ *(r + (f->offset)) = 42; \]
No type checking of f.

CHERI JNI checks f and r are sealed with the correct type, errors if not.

// Get the field ID for integer field x
jfieldID f = (*env)->GetFieldID(env, cls, "x", "I");

// Set that field to 42
(*env)->SetIntField(env, r, f, 42);
### CHERI vs prior sandboxing work

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>JITs</th>
<th>Stack Unwinders</th>
<th>Many Sandboxes</th>
<th>Direct buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHERI</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>SFI-based</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Process-based</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Garbage collection extends to C

• All Java references in C code are sealed capabilities

• All pointers to Java arrays or direct buffers are unsealed (but bounded) capabilities

• All capabilities are protected by a tag bit

• The garbage collector can find them in memory
Conclusion

• CHERI allows the Java security model to be extended all of the way through native code

• Native code cannot violate the invariants of the JVM

• Performance is comparable with conventional JNI implementations