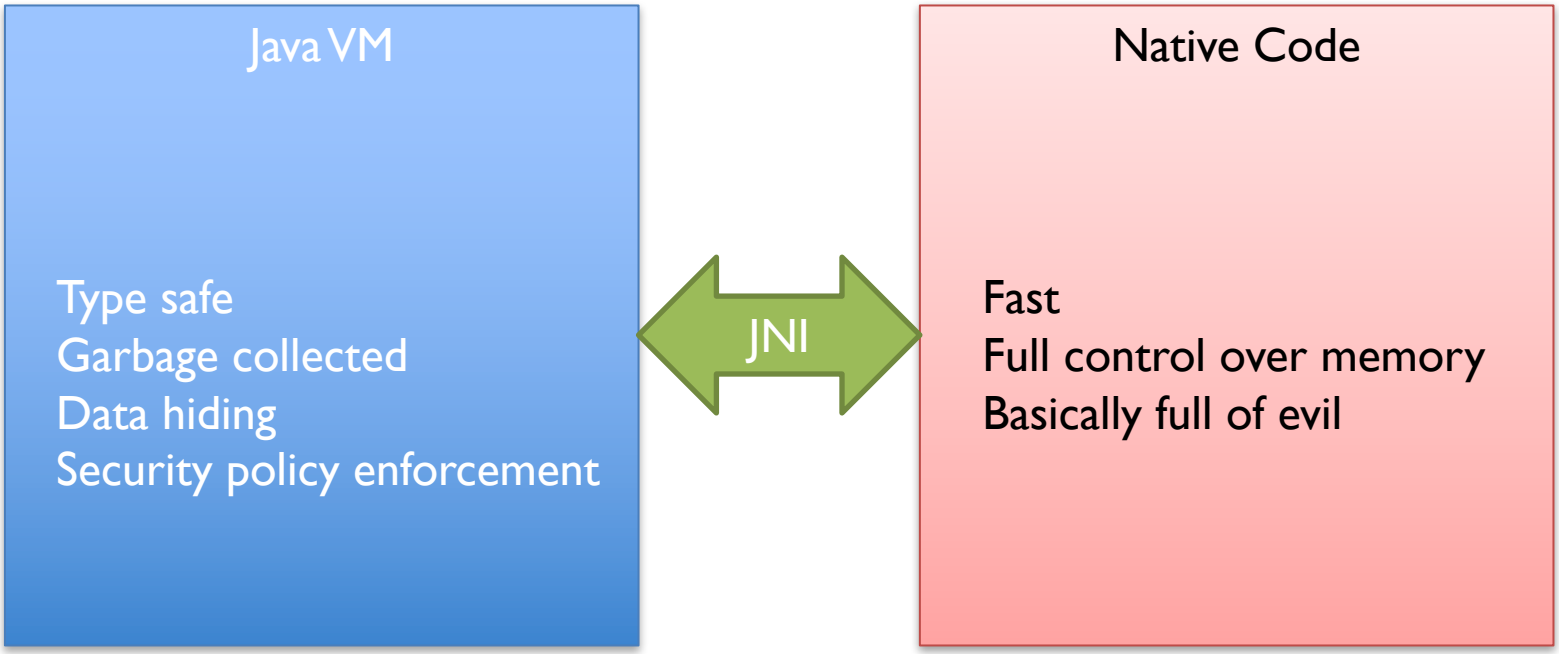
# CHERI JNI:

## Sinking the Java security model into the C

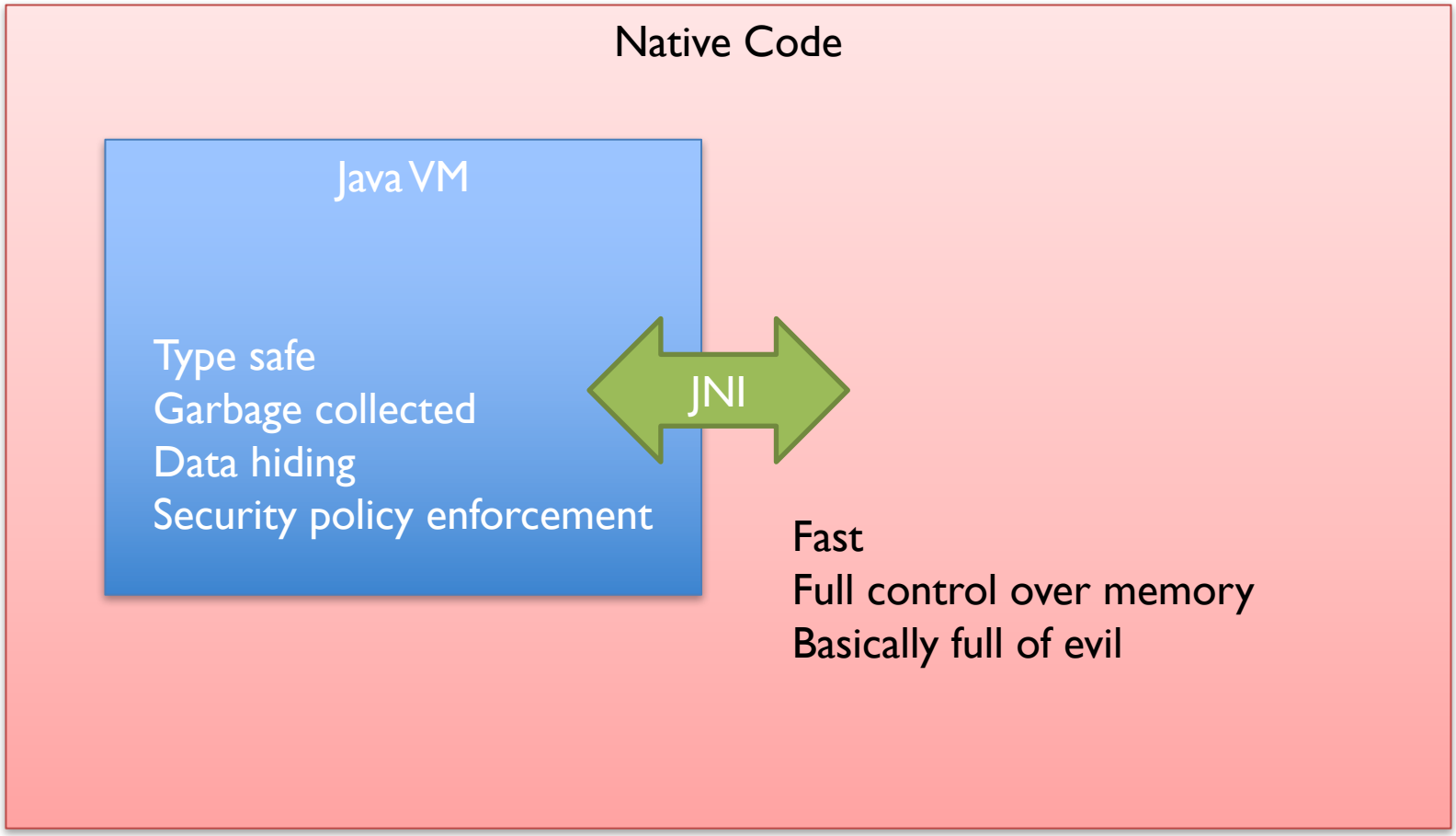
**David Chisnall**, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, Robert N. M. Watson

Approved for public release; distribution is unlimited. This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

# The Java abstract machine



# The Java concrete machine

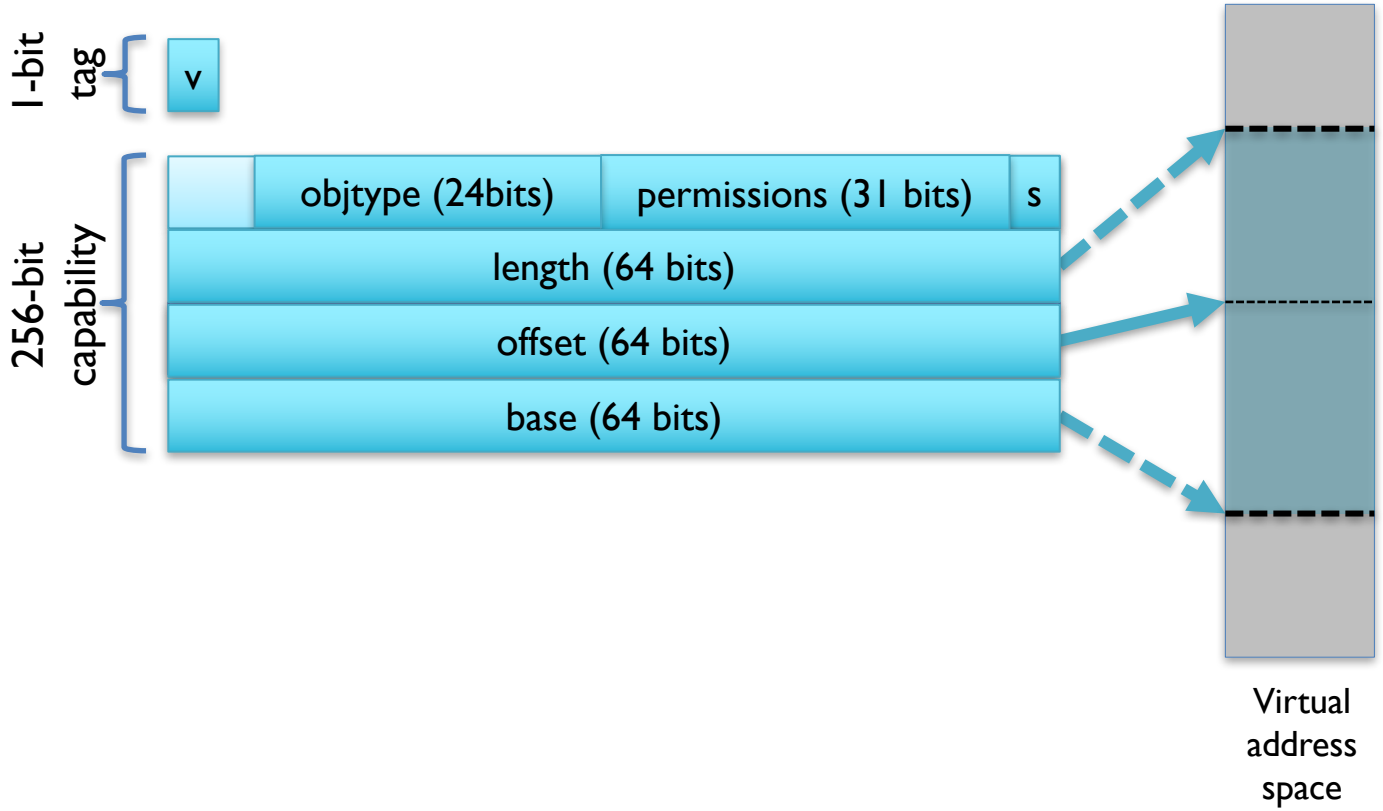


# JNI is insecure by design

*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.*

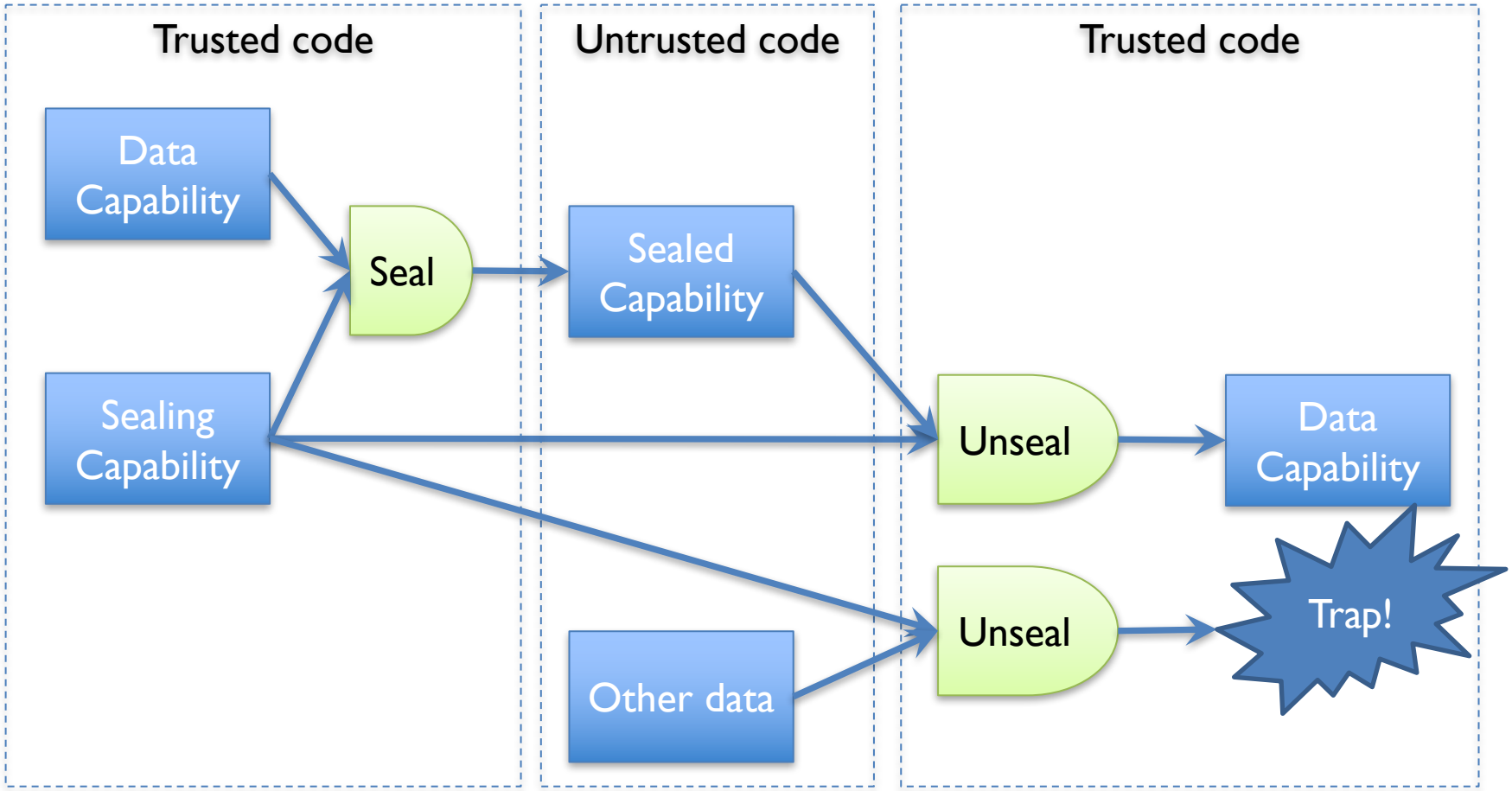
# BRIEF CHERI PRIMER

# Hardware memory safety



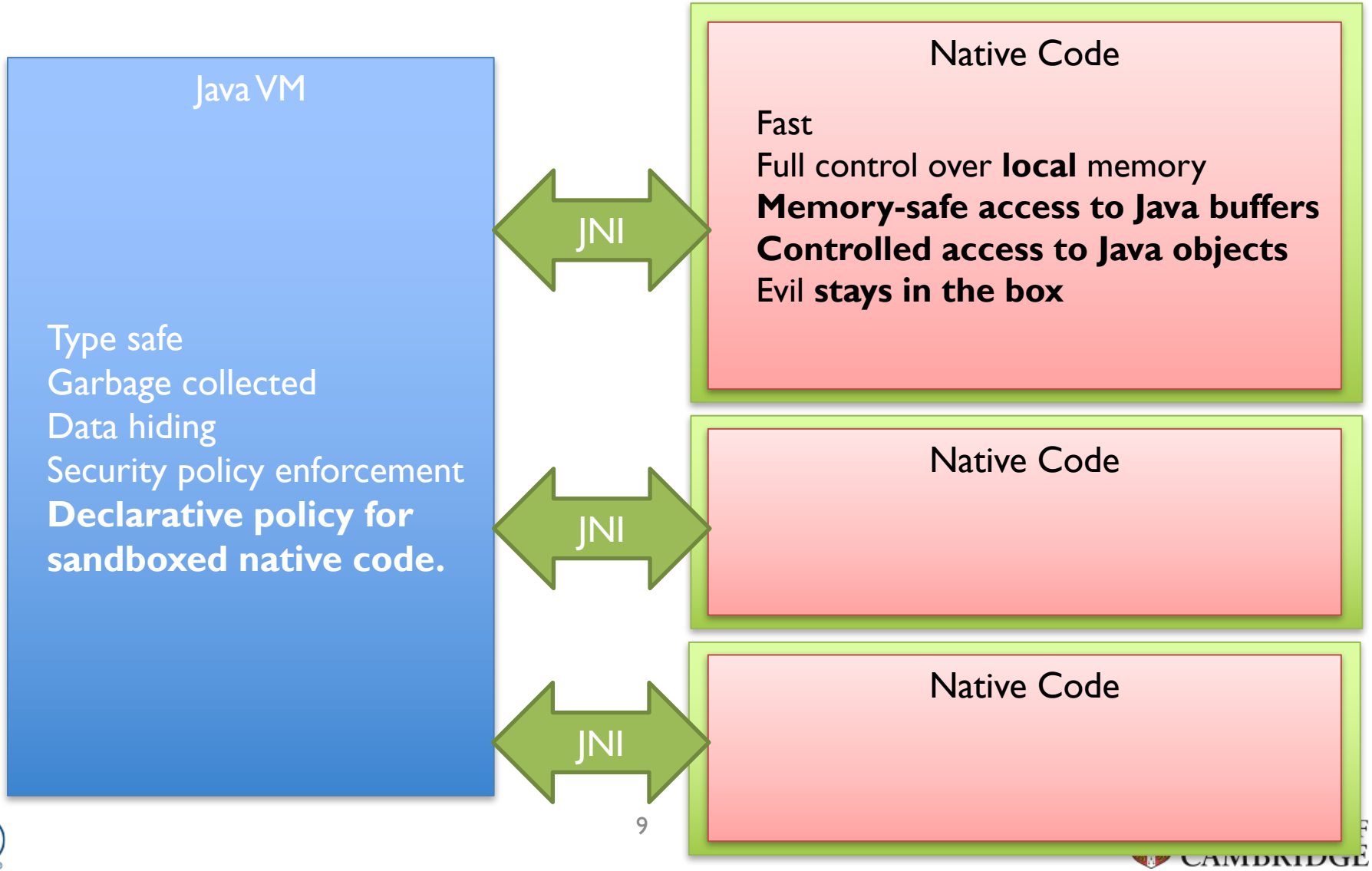

Compressed representation provides the same abstract model in 128 bits.

# Sealing gives opaque pointers

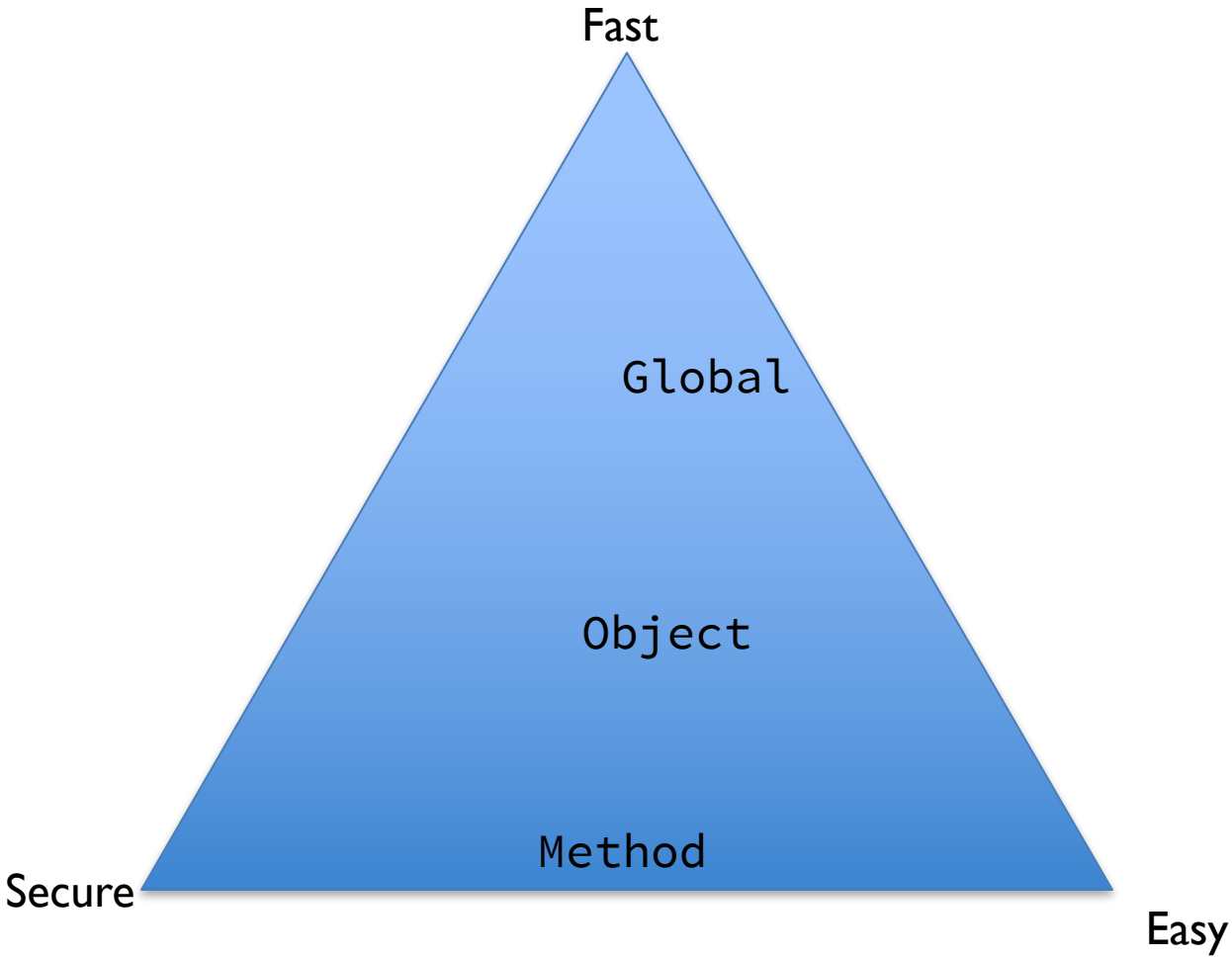


# CHERI AND THE JNI

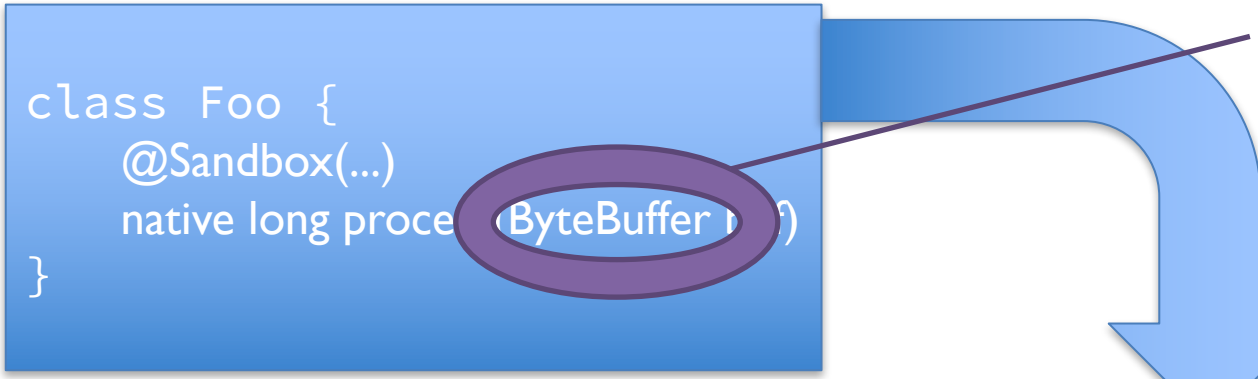
# CHERI JNI


Java VM
Type safe
Garbage collected
Data hiding
Security policy enforcement
Declarative policy for sandboxed native code.
JNI
JNI
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
Native Code
Native Code

# Sandbox scopes



# Direct buffer access




Java NIO class intended to provide C code with direct access to JVM-owned memory

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

# Direct buffer access

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

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

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

# 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

| Mechanism | JITs | Stack Unwinders | Many Sandboxes | Direct buffers |
|---|---|---|---|---|
| CHERI | ✔ | ✔ | ✔ | ✔ |
| SFI-based | X | X | X | X |
| Process-based | ✔ | ✔ | X | X |

# 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