UT Austin Computer Science Technical Report TR-09-39. Under double-blind submission to PLDI 2010.

Jinn: Synthesizing a Dynamic Bug Detector for Foreign Language Interfaces [*]

Byeongcheol Lee
University of Texas at Austin
bclee@cs.utexas.edu

Martin Hirzel
IBM Watson Research Center
hirzel@us.ibm.com

Robert Grimm
New York University
rgrimm@cs.nyu.edu

Ben Wiedermann Kathryn S. McKinley
University of Texas at Austin
{ben,mckinley}@cs.utexas.edu

Abstract

Programming language specifications mandate static and dynamic analyses to preclude syntactic and semantic errors. Although individual languages are usually well-specified, composing languages in *multilingual* programs is not. Because multilingual programs are prevalent, poor specification is a source of many errors. For example, virtually all Java programs compose Java and C with the Java Native Interface (JNI). Unfortunately, JNI is informally specified and thus, Java compilers and virtual machines (VMs) check only a small subset of JNI constraints. Worse, Java compiler and VM implementations inconsistently check constraints.

This paper's most significant contribution is to show how to synthesize dynamic analyses from state machines to detect foreign function interface (FFI) violations. To demonstrate the generality of our approach, we build FFI state machines that encode specifications for JNI and Python/C. Although we identify over a thousand FFI correctness constraints, we show that they fall into three classes and a modest number of state machines encode them. From these state machines, we generate context-specific FFI dynamic analysis. For Java, we insert this analysis in a library that interposes on all language transitions and thus is compiler and VM *independent*. We call the resulting dynamic bug detection tool *Jinn*. We show Jinn detects and diagnoses a wide variety of FFI bugs that other tools do not. This paper lays the foundation for better specification and enforcement of FFIs and a more principled approach to developing correct multilingual software.

1. Introduction

Programming language designers often spend years on precisely specifying their languages, including formalizing and standardizing them [9, 12, 19]. Likewise language implementors exert considerable effort towards enforcing these specifications through static and dynamic checks. Many developers, however, compose multiple languages to reuse legacy code and leverage the languages best suited to their needs. Such multilingual programs require additional specification, i.e., a *foreign function interface* (FFI) that defines how languages interoperate in the presence of syntactic and semantic differences. Well-designed and well-specified FFIs include recent integrated language designs [11, 20] and language binding synthesizers [4, 18], but programmers have not yet widely adopted these approaches, in part, because it requires re-writing programs.

The FFIs currently in wide use, such as the Java Native Interface (JNI) and Python/C, are large, under-specified, and hard to use correctly. They have hundreds of API calls, each with many complex usage rules. For example, while JNI is well-encapsulated and portable, it has 229 API calls and over 1,500 usage rules. Much of this complexity is due to the impedance mismatch between languages, e.g., differences in object models, memory management, and exceptions. Example API calls include: look up a class by name and return its class descriptor; look up a method by class descriptor and signature and return a method descriptor; and invoke a method by its descriptor. Usage rules include: when a program calls from C to Java, it must not have a pending Java exception; and when a program needs more than 16 local cross-language pointers, it must make an extra request [16].

Voluminous, complex, and context-dependent FFI rules lead to three problems. (1) Many rules prevent sound and complete static FFI usage validation and thus only dynamic checking can validate them. (2) Complex under-specified rules lead to inconsistent language implementations. Even if different implementations enforce a given rule, they produce divergent results in some cases, such as signaling an exception versus terminating the program. (3) Writing correct FFI code that follows the rules is hard for programmers. As a direct consequence of these three problems, real-world multilingual programs are full of FFI bugs [7, 8, 13, 14, 15, 21, 22].

This paper significantly improves on this sorry state by presenting a systematic and practical approach to dynamic FFI validation. We first demonstrate that JNI's 1,500+ usage rules boil down to the following three classes of rules, parameterized by the FFI methods and their arguments:

JVM state constraints: restrictions on JVM state that require a specific JVM thread context, critical section state, and/or exception state.

Type constraints: restrictions on parameter types, parameter values (e.g., NULL or not NULL), and semantics (e.g., no writing to a final field).

Resource constraints: restrictions on the number of multilingual pointers and on the lifetime of resources such as locks, metadata, and memory.

We then show how to express these rule classes with state machines, which precisely capture illegal FFI usage.

The most significant contribution of this paper is the design, implementation, and demonstration of a dynamic analysis synthesizer called Jinn that takes these state machines and generates a context-sensitive dynamic analysis that enforces the FFI specification. We dynamically and transparently inject the analysis into user code

[*] This work is supported by NSF SHF-0910818, NSF CSR-0917191, NSF CCF-0811524, NSF CNS-0719966, NSF CCF-0429859, Samsung Foundation of Culture, Intel, IBM, CISCO, Google, and Microsoft. Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

	Production runs		JNI error detection in prior work				
					Dynamic		
JNI Pitfall	HotSpot	J9	Language	Static	HotSpot	J9	Jinn
1. Error checking	running	**crash**	[11], [20]	[13], [15]	warning	error	exception
2. Invalid arguments to JNI functions	running	**crash**	[11], [20]	[8], [22]	running	**crash**	exception
3. Confusing jclass with jobject	**crash**	**crash**	[11], [20]	[8]	error	error	exception
6. Confusing IDs with references	**crash**	**crash**	[11]	[8]	error	error	exception
8. Terminating unicode strings	**crash**	running	[11], [20]		**crash**	running	**crash**, running
9. Violating access control rules	NPE	NPE	[11], [20]		NPE	NPE	exception
11. Retaining virtual machine resources	leak	leak	[11], [20]	[13]	**crash**	error	exception
12. Excessive local reference creation	leak	leak			running	error	exception
13. Using invalid local references	**crash**	**crash**	[11]	[13]	error	error	exception
14. Using the JNIEnv across threads	running	**crash**	[11]		error	**crash**	exception
16. Bad critical region	deadlock	deadlock	[11]	[13]	running	warning	exception

Table 1. JNI pitfalls and effects. *Running*: continues to execute in spite of undefined JVM state. *Warning*: prints a warning message, and keeps running. *Error*: prints a warning message and aborts. *NPE*: raises a null pointer exception. *Exception*: raises a Jinn JNI failure exception.

by using language interposition implemented with vendor-neutral JVM interfaces. The JVM loads Jinn together with the program during start-up, and then Jinn interposes on all Java and C transitions. To the JVM, Jinn looks like normal user code, whereas to user code, Jinn is invisible. Jinn checks FFI constraints at every language transition, thus diagnosing bugs when the program actually violates an FFI constraint.

Jinn is more practical than the state-of-the-art in finding JNI bugs, in part because it does not depend on the JVM or C compiler. Furthermore, Jinn is complementary to static analysis, because it has no false positives and finds only exercised bugs. At the same time, static tools cannot conclusively identify some dynamic resource bugs that Jinn does find. The experimental results section shows that Jinn works with unmodified programs and VMs, incurs only a modest overhead, and that programmers can examine the full program state when Jinn detects a bug.

While we explore JNI in depth, we show that the Python/C FFI has similar characteristics: three rule classes encoded in state machines express hundreds of FFI usage rules. These rule classes stem from a set of fundamental language semantic mismatches: whereas C relies on manual resource management and is only weakly typed, higher-level languages like Java and Python automate memory reclamation, provide a stronger typing discipline (whether static or dynamic), enforce well-formed resource access, and automate error propagation. We implement and evaluate a bug detector that checks a subset of the FFI rules we identified for Python/C to provide further evidence for the generality of our approach.

The contributions of this paper include:

- Synthesis of dynamic analysis from state machines that rigorously specify a foreign function interface.
- State machine specifications for JNI based on its informal specification [16].
- Jinn, a tool generated by our synthesizer, which is the most practical JNI bug detector to date.
- A demonstration of Jinn on JNI microbenchmarks and on real-world programs.
- A demonstration of the generality of this approach for Python/C.

We believe that by identifying and capturing the three classes of FFI constraints, this paper helps lay a foundation for a more principled approach to designing foreign function interfaces and to developing correct multilingual software.

2. Motivation and Related Work

This section first shows that JNI has inconsistent implementations and behaviors, which likely stem from poor specification and certainly complicate portable JNI programming. We then quantitatively and qualitatively compare Jinn to prior work that uses language design, static analysis, and dynamic analysis to find and report FFI bugs.

Programming against an FFI is challenging because programmers must reason about multiple languages and the interactions of their semantics. For example, Chapter 10 of the JNI manual identifies a number of pitfalls [16]. We list the most serious of these in Table 1, using Liang's numbering scheme, and also include "bad critical region" from Chapter 3.2.5 as a 16th pitfall. We created microbenchmarks to exercise these pitfalls and executed that code with HotSpot and J9. Columns two and three show the results: JNI mistakes cause a wide variety of crashes and silent corruption. Furthermore, the two VMs show different behavior on four of the ten errors. As shown in columns six and seven, the VMs do not do much better when we turn on built-in JNI checking with the `-Xcheck:jni` command-line flag.

Table 1 also compares language designs, static analysis tools, and our Jinn implementation. An empty entry indicates that we are not aware of a language feature or static analysis that handles this pitfall. We fill in entries based on our reading of the literature [8, 11, 13, 15, 20, 22]. we did not execute the static tools or re-write our microbenchmarks. Language designs cover the widest class of JNI bugs [11, 20], but new languages require developers to rewrite their code. Static analysis can catch some of these pitfalls, but, in general, dynamic and static analysis are complimentary: dynamic analysis misses unexercised bugs, whereas static analysis reports false positives.

The last column shows that Jinn detects nine of ten serious and common errors. Error 8 does not actually violate JNI usage rules, but may cause memory corruption in C code and is platform dependent. When Jinn detects any of the other errors, it throws a JNI failure exception and stops execution to help programmers debug. Jinn works out-of-the-box on unmodified JNI, which makes it practical for use on many existing programs. It systematically finds more errors than all the other approaches.

2.1 Language Approaches to FFI Safety

Two language designs propose to replace the JNI. SafeJNI [20] combines Java with CCured [17], and Jeannie safely and directly nests Java and C code into each other using quasi-quoting [11]. Both SafeJNI and Jeannie define their language semantics such that static checks catch many errors and both add dynamic checks in translated code for other errors. From a purist perspective, preventing FFI bugs while writing code is more satisfying than spending time to fix them after the fact. Another approach generates language bindings for annotated C and C++ header files [4]. Ravitch et al. reduce the annotation burden for this approach [18]. Jinn is more practical than these approaches, because it does not require developers to annotate or rewrite their code in a different language.

2.2 Static FFI Bug Checkers

A variety of static analyses verify foreign function interfaces [7, 8, 13, 15, 21, 22]. Static analysis frameworks that use finite state machines (FSMs) to express specifications [6, 10] are similar in spirit to Jinn: whereas they generate static analysis, we generate dynamic analysis from more general constraints. All static analysis approaches suffer from false positives, whereas Jinn will never generate false positives. On the other hand, dynamic analysis only finds bugs when they actually occur, so Jinn is complementary to these static analyses. Yet Table 1 shows that Jinn finds strictly more JNI bugs than the static analyzers. Furthermore, static analysis requires access to the complete C source code of native libraries, whereas Jinn requires no source code access. Finally, Jinn accepts all native languages. For instance, Jinn found FFI bugs in the Subversion Java binding written in C++, which is currently not handled by static JNI bug-finders [8, 13, 15].

2.3 Dynamic FFI Bug Checkers

Some JVMs provide built-in dynamic JNI bug checkers, enabled by the `-Xcheck:jni` command-line flag, and print warning messages when they detect incorrect JNI usage. While convenient, these error checkers only cover limited classes of bugs, and the JVMs of different vendors implement them inconsistently. The Blink debugger provides JNI bug checkers that work consistently for different JVMs, but the coverage of checked bugs is limited to only two classes: validating exception state and nullness constraints [14]. These two kinds of checks are easy to implement, because they require no preparatory bookkeeping.

Jinn covers a larger class of JNI bugs, works consistently with any JVM that implements the JVMTI, and explicitly throws an exception at the point of failure. An FFI specification must be enforced consistently across language implementations to support a principled approach to software quality. For instance, a graphical user interface program cannot display messages for a JNI bug from the JVM built-in checkers. In contrast, Jinn throws an exception for the JNI bug, and if desired, the program can report the exception in a dialog. When the error message and calling context from the exception do not suffice to find the cause of the failure, programmers may rerun the program with Jinn and a Java debugger. The Java debugger catches the exception, and the programmer can access the program state at the point of failure.

We are the first to synthesize a bug detector for an FFI from a specification, but prior work has synthesized other dynamic analyses from specifications. Allan et al. turn FSMs into dynamic analyses by using aspect-oriented programming [2]. Chen and Rosu synthesize dynamic analyses from a variety of specification formalisms, including FSMs [5]. And Arnold et al. implement FSMs for bug detection in a JVM, and control the runtime overhead by sampling [3]. Like these approaches, Jinn turns specifications in the form of more general state machines into a dynamic bug detector. Whereas the previous work can only handle a single language at a time, Jinn focuses on catching multilingual bugs.

3. Synthesizing FFI Bug Detectors

This section starts with some JNI background and an example JNI bug to motivate our use of state machines for JNI specification.

```
1.  JNIEXPORT void JNICALL Java_Callback_bind(JNIEnv *env,
2.     jclass clazz, jclass receiver, jstring name, jstring desc)
3.  {              /* Register an event call-back to a Java listener. */
4.     EventCallBack* cb = create_event_callback();
5.     cb->handler = callback;
6.     cb->receiver = receiver;      /* receiver is a local reference.*/
7.     cb->mid = find_java_method(env, receiver, name, desc);
8.     if (cb->mid != NULL) enable_callback(cb);
9.     else destroy_callback(cb);
10. }                                  /* receiver is a dead reference. */
11. static void callback(EventCallBack* cb, Event* event) {
12.    JNIEnv* env = find_env_pointer_from_current_thread();
13.    jvalue* jargs = marshal_event(cb, env, event);
14.    /* Here is a bug: the invalid local reference of cb->receiver. */
15.    (*env)->CallStaticVoidMethodA(
16.       env, cb->receiver, cb->mid, jargs);
17. }
```

Figure 1. JNI invalid local reference error in a call-back routine from GNOME (Bug 576111) [23].

Section 3.2 then describes how to synthesize a dynamic FFI bug detector from this specification, and Section 4 presents our JNI constraint classification, descriptions of individual constraints, and their representations as state machine specifications. We use state machines, instead of finite state machines, to enforce constraints that need to take into account resources and calling context.

The JNI is designed to hide JVM implementation details from native code, while also supporting high-performance native code. Hiding JVM details from C code makes multilingual Java and C programs portable across JVMs and gives JVM vendors flexibility in memory layout and optimizations. Achieving portability together with high performance leads to 229 API functions and over 1,500 usage rules. For instance, JNI has functions for calling Java methods, accessing fields of Java objects, obtaining a pointer into a Java array, and many more. To hide JVM implementation details, these functions go through an indirection, such as method and field IDs, or require pinning arrays by the garbage collector. Developers using JNI avoid much of the indirection overhead on the C side by caching method and field IDs, by distinguishing thread-local from global references, and by manually releasing pinned resources.

3.1 Example FFI Bug

Figure 1 shows a simplified version of an FFI bug from the GNOME project Bugzilla database (Bug 576111) [23]. We use this bug to illustrate how programmers violate low-level FFI constraints and to motivate our use of state machines for rigorous specification. GNOME is a graphical user interface that makes heavy use of several C libraries, and it has a variety of language bindings for C++, Java, Ruby, C#, and Python. Line 1 defines a C function Java_Callback_bind that implements a Java native method using the JNI. An example *call from Java to C* takes the following form:

```
Callback.bind(receiverClass, "methodName", "description");
```

This call invokes the C function Java_Callback_bind, which allocates a new C heap object cb storing the receiver class and method name passed as parameters from Java. Line 11 defines a C function callback that uses the cb object to call from C code to the specified Java method. Line 15 shows this *call from C to Java*. It uses a JNI API function CallStaticVoidMethodA, residing in a struct referenced by the JNI environment pointer env.

This code is buggy. The parameter receiver in Line 2 is a local reference. A local reference in JNI is only valid until the enclosing function returns, because, otherwise, it might inhibit garbage collection. In this case, the receiver becomes invalid when the function returns at Line 10. However, the receiver escapes from the function when Line 6 stores it in a heap object. Line 16 retrieves receiver

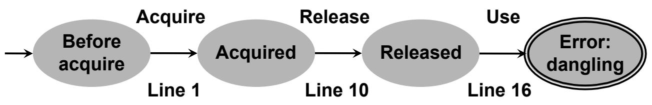


Figure 2. A state machine to track the reference error in Figure 1.

from the heap and uses it as a parameter to CallStaticVoidMethodA. At this point, receiver is a dangling reference, and the JVM may already have reclaimed and reused it. Using receiver produces unspecified results. The JNI specification merely says that it is invalid, but leaves the consequences up to the vendor's Java implementation [16]. This kind of bug is difficult to find with static analysis, because it involves complex data flow through the heap, and complex control flow through disjoint indirect calls and returns across languages. For instance, the syntax analysis in J-BEAM [13] misses this bug, and the static type and data-flow analyses in Table 1 generate hundreds and thousands of false alarms.

The state machine specification in Figure 2 describes how using a reference after release leads to an error. When control enters the function in Line 1, the state machine for the receiver object transitions to the Acquired state. When control leaves the function in Line 10, the state machine transitions from the Acquired state to the Released state. Jinn dynamically performs these state transitions on each method invocation for the actual parameter. If, at some later time, Line 16 uses receiver, the state machine transitions from the Released state to the Error: dangling state. Other resource usage state machines follow this pattern and are described in Section 4.

3.2 Dynamic Analysis Synthesis

We use state machine specifications like the one in Figure 2 to synthesize a dynamic analysis. Each state machine specification may describe several state transitions, which in turn may be triggered by any number of language transitions. The cross-product of these possibilities yields thousands of checks in the dynamic analysis. For example, before Line 15 in Figure 1, the analysis must ensure at least eight constraints:

- The Java interface pointer, env, matches the current C thread.
- The current JVM thread does not have pending exceptions.
- The current JVM thread does not have any open critical and direct access to any Java objects.
- cb->mid is not NULL.
- cb->receiver is not NULL.
- cb->receiver is not a dangling JNI reference.
- cb->receiver is a reference to a Java Class object.
- The formal arguments of cb->mid are compatible with the actual arguments in cb->receiver and jargs.

Hand-coding all these constraints would be tedious and error-prone. Instead, we designed a system that synthesizes a dynamic analysis from state machine specifications. The specification of a single state machine includes three components:

State Representation: Information about the program execution that represents the current state of the state machine. For example, Jinn represents the Acquired state of Figure 2 by recording the reference in an analysis-internal thread-local list.

Trigger Checks: Code that checks whether a state machine transition triggers. Each state transition occurs at a language boundary. For example, when Java calls function `Java_Callback_bind` in Figure 1, the state machine instance for each argument transitions from the Before acquire state to the Acquired state. When the method exits and control returns to Java, each argument transitions from the Acquired state to the Released state. An important special case is the trigger check for transitions to an error state. For example, the call to JNI function `CallStaticVoidMethodA` at Lines 15–16 in Figure 1 causes the state machine in Figure 2 to transition to the Error: dangling state, because the `cb->receiver` reference has been released.

State Transformers: Code that changes the state representation to encode a state machine's transition to a new state. For example, when the state machine in Figure 2 transitions from Before acquired to Acquired, the state transformer inserts the reference into the list of acquired references.

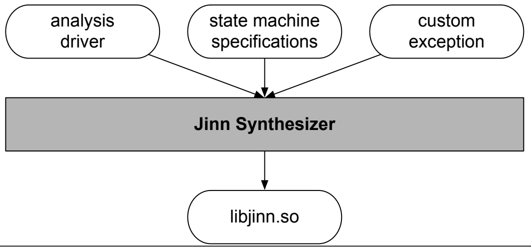


Figure 3. Structure of Jinn Synthesis.

To synthesize an analysis from a collection of state machines, the programmer writes code that expresses the three components of each state machine specification. This code serves as input to the following algorithm:

Algorithm 1 Given state machine specifications, synthesize a dynamic analysis by augmenting FFI functions.

```
1: for each state machine specification $M$ do
2:     for each state transition $s_a \rightarrow s_b \in M$ do
3:         for each FFI function $f$ that triggers $s_a \rightarrow s_b$ do
4:             augment $f$ with the following synthesized code:
5:             for each instance $m$ of $M$ do
6:                 if the [Trigger Check] for $(s_a \rightarrow s_b)$ succeeds then
7:                     execute the [State Transformer] code to change
                       the [State Representation] of $m$ from $s_a$ to $s_b$.
```

The algorithm first computes the cross product of machine transitions and FFI functions, then generates a wrapper for each FFI function that performs the appropriate state transformations and error checking. This functionality is the core of the Jinn Synthesizer component in Figure 3. The synthesizer takes two additional inputs: an analysis driver and a custom exception. The analysis driver initializes the state data and dynamically injects the generated, wrapped FFI functions into a running program. The custom exception defines how the dynamic analysis reports errors.

The output of the synthesizer is Jinn—a shared object file that the JVM dynamically loads using the JVM tools interface (JVMTI). Jinn monitors runtime events and program state. When Jinn detects a bug, it throws the custom exception. If the exception is not handled, the JVM prints a message with the JNI constraint violation and the faulting JNI function call. If Jinn is invoked within a debugger, the programmer can inspect the call chain, program state, and other potential causes of the failure.

Our discussion now turns to the three classes of JNI constraints that Jinn monitors. These three classes collectively can be described with nine state machine specifications, which encompass the more than 1,500 usage rules described in the JNI manual.

4. JNI Constraint Classification

We classify JNI constraints into three classes. (1) JVM state constraints ensure that the JVM is in the right state before calls from C.

Constraint	Count	Description
JVM state constraints		
`JNIEnv*` state	229	Current thread matches `JNIEnv*` thread.
Exception state	209	No exception pending for sensitive call.
Critical-section state	225	No critical section open for sensitive call.
Type constraints		
Nullness	416	Parameter is not null.
Fixed typing	145	Parameter matches API function signature.
Entity-specific typing	130	Parameter matches Java entity signature.
Access control	54	Written field is non-final.
Resource constraints		
Manually-managed resource	11	No overflow, leak, dangling reference, or double-free.
Semi-automatic resource	269	No overflow, leak, dangling reference, or double-free.

Table 2. Classification and Number of JNI constraints.

(2) Type constraints ensure that C passes valid arguments to Java. (3) Resource constraints ensure that C code manages JNI resources correctly. Table 2 summarizes these constraints and indicates the number of times Jinn's language interposition agent checks them. For example, the "`JNIEnv*` state" constraint appears 229 times, because Jinn checks its validity in all 229 JNI functions. The remainder of this section discusses all these constraints in detail.

4.1 JVM State Constraints

To enter the JVM through any JNI function, C code must satisfy three conditions. (1) The JNI environment pointer `JNIEnv*` and the caller belong to the same thread. (2) Either no exception is pending, or the callee is exception-oblivious. (3) Either no critical region is active, or the callee is critical-region oblivious.

JNIEnv* thread constraints. All calls from Java to C implicitly pass a pointer to the `JNIEnv` structure, which specifies the JVM-internal and thread-local state. All calls from C to Java must explicitly pass the correct pointer when invoking a JNI function. Jinn tracks `JNIEnv` pointers by associating the native thread ID with its `JNIEnv*` on thread creation and discarding that mapping on thread termination. For each JNI call, Jinn looks up the expected `JNIEnv*` based on the thread's ID and compares it to the actual pointer, reporting a bug if the pointers differ. We summarize this constraint checking as follows.

JNIEnv* State: Current thread matches JNI* thread.
State: Map from thread IDs to expected `JNIEnv*` pointers.
Check: Verify actual `JNIEnv*` matches expected pointer.

JVM exception constraints. When Java code throws an exception and returns to C, the C code does not automatically transfer control to the nearest exception handler: the program must explicitly consume or propagate the pending exception. This constraint results from the semantic mismatch in how C and Java handle exceptions.

To consume a pending exception, the programmer selects from four JNI functions that query and clear the pending exception. To propagate the pending exception to Java code, the programmer selects from 16 JNI functions that release JVM resources. All these 20 JNI functions can be called while an exception is pending in the current thread (they are exception-oblivious). The programmer must not call any of the remaining 209 functions (they are exception-sensitive). Validating this constraint requires the following state and actions.

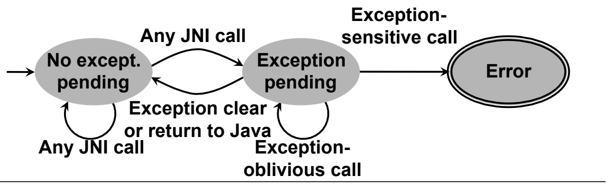
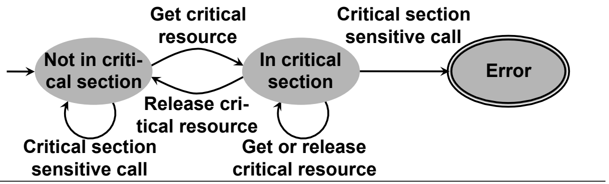


Figure 4. Exception states in JNI.



Get critical resource
Critical section sensitive call
Not in critical section
In critical section
Error
Release critical resource
Critical section sensitive call
Get or release critical resource

Figure 5. Critical section states in JNI.

Exception state: No exception is pending upon entry to an exception-sensitive call.
State: Map from thread IDs to pending Java exceptions.
Check: Verify current thread has no pending exception.

Figure 4 illustrates the JNI exception states. When Java calls C code, no exception is pending in the current thread. Any JNI function may or may not throw an exception, transitioning to the "exception pending" state. The JVM internally records this state transition for each Java thread, so Jinn does not need to perform any bookkeeping at this point. At each call to an exception-sensitive function, Jinn calls `ExceptionCheck` to check whether an exception is pending. This call is exception-oblivious. If an exception is pending, Jinn wraps that exception in an error report to the user.

JVM critical section constraints. JNI gives C code direct access to the memory underlying a string or array on demand (with two functions: `GetStringCritical` and `GetPrimitiveArrayCritical`). The C code should subsequently release the critical resource (with `ReleaseStringCritical` and `ReleasePrimitiveArrayCritical`). The C code is expected to hold these resources for a short time, because the JVM may implement this feature by disabling the garbage collector to prevent the objects from moving. These four functions are thus considered critical-section *insensitive* and the remaining 225 JNI function calls are critical-section *sensitive*. To prevent deadlock, the C code may not call JNI sensitive functions if the current thread has an outstanding "Get" without a matching "Release."

Figure 5 illustrates JNI critical-section state tracking. Jinn instruments the four "Get" and "Release" calls to increment and decrement a thread-local counter. When the counter crosses the threshold between 0 and 1, the state transitions between non-critical and critical. Jinn checks that the state is zero before each call to any of the 225 critical-section sensitive functions.

Critical-section state: No critical section is open when calling a JNI critical-section sensitive function.
State: Map from thread IDs to critical-section counts.
Check: Verify critical-section count is zero.

4.2 Type Constraints

When Java code calls a Java method, the compiler and JVM check type constraints on the parameters. But when C code calls a Java method, the compiler and JVM do not check type constraints, and type violations cause unspecified JVM behavior. For example, given the Java code: `Collections.sort(ls, cmp);` the Java compiler checks that class `Collections` has a static method `sort`, and that the actual parameters `ls` and `cmp` conform to the formal parameters of `sort`. Given the equivalent code expressed with Java reflection:

```
Class clazz = Collections.class;
Method method = clazz.getMethod(
                "sort", List.class, Comparator.class);
method.invoke(Collections.class, ls, cmp);
```

the Java compiler cannot statically verify the safety, but the JVM throws an exception at runtime to maintain safety. With JNI, this code reads:

```
jclass clazz = (*env)->FindClass(env,
                          "java/util/Collections");
jmethodID method=(*env)->GetStaticMethodID(env, clazz,
  "sort","(Ljava/lang/List;Ljava/util/Comparator;)V");
(*env)->CallStaticVoidMethod(env,clazz,method,ls,cmp);
```

The compiler and JVM do not check *any* constraints on `Collections`, `sort`, `ls`, or `cmp`. Furthermore, there are additional JNI-specific parameters `env`, `clazz`, and `method` with their own type constraints, which the compiler and JVM do not check either. This section details the type constraints and how Jinn checks them.

Nullness. Some parameters are not allowed to be null. For example, parameters `env`, `clazz`, and `method` in `CallStaticVoidMethod(env, clazz, method, ls, cmp)` must not be null. Some JNI functions accept null parameters, for example, for the initial array elements in `NewObjectArray`. Since the JNI specification is not always clear on which parameters may be null, we determined these constraints experimentally. In the end, we found 416 non-null constraints in various JNI functions. Nullness checking requires no preparatory Jinn bookkeeping.

Nullness: Parameter is not null.
State: No state.
Check: Verify actual parameters are not NULL.

Legal types. Type constraints require the runtime type of actuals to conform to the formals. In the simple case, the type is fixed by the JNI function. For example, in `CallStaticVoidMethod(env, clazz, method, ls, cmp)`, the `clazz` actual must always conform to type `java.lang.Class`, irrespective of the value of `method`. We carefully reviewed the JNI specification to determine all the "fixed" type constraints, i.e., where the C parameter declaration in the specification (such as `jstring`) has a well-defined corresponding Java type (e.g., `java.lang.String`). While most functions have such fixed type constraints, JNI also contains 13 functions with type constraints that are only specified in the explanation. For instance, `FromReflectedMethod` has a `jobject` parameter, whose expected type is either `java.lang.reflect.Method` or `java.lang.-reflect.Constructor`. To check all type constraints, Jinn obtains the class of the actual using `GetObjectType` and then checks compatibility with the expected type through `IsAssignableFrom`.

Fixed typing: Parameter matches API function signature.
State: No state.
Check: Verify types of actuals conform to formal parameters.

JNI refers to Java fields and methods via IDs. For example, in `CallStaticVoidMethod(env, clazz, method, ls, cmp)`, parameter `method` is a method ID. In this case, the method must be static, and the method parameter determines constraints on the other parameters. In particular, the `clazz` must declare the method, and `ls` and `cmp` must conform to the formal parameters of the method.

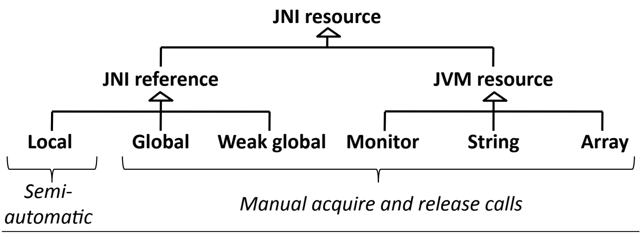


Figure 6. Classification of JNI resources.

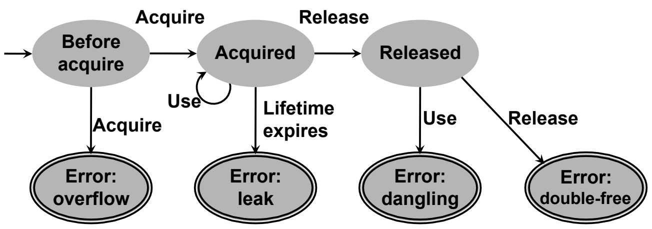


Figure 7. Resource states.

Entity-specific typing: Parameter matches Java field or method signature.
State: Map from member IDs to their signatures.
Check: Verify actual parameters match member IDs.

A plethora of JNI functions call Java methods or access Java fields. In all cases, the entity identifier constrains types of method parameters or field values, the receiver class (for static entities) or object (for instance entities), and the proper JNI function to use. Jinn records method and field signatures upon return from JNI functions that produce method and field IDs. Jinn checks all these IDs conform when a JNI function uses them.

Access control. Even when type constraints are satisfied, Java semantics may prohibit accesses based on visibility and `final` modifiers. For example, in `SetStaticIntField(env, clazz, fid, 42)`, the field identified by `fid` may be private or final, in which case the assignment follows questionable coding practices. It is not immediately clear exactly which accesses should be allowed from JNI based on visibility and `final` constraints. After some investigation, we found that JNI usually ignores visibility, but honors the `final` modifier. Ignoring visibility rules seems surprising, but as it turns out, this permissiveness is consistent with the behavior of reflection when `setAccessible(true)` was successful. Honoring `final` is common sense: despite the fact that reflection can also mutate final fields, doing so interferes with JIT optimizations and concurrency, and complicates the Java memory model. Jinn thus warns against it. Jinn records whether each field is final upon return from JNI functions that produce field IDs. Jinn checks that JNI does not attempt to write final fields.

Access control: Written field is non-final.
State: Map field IDs and access modifiers.
Check: Verify destination field is not final in calls to `Set<Type>Field` and `SetStatic<Type>Field`.

4.3 Resource Constraints

Figure 6 shows the six kinds of JNI resources. A JNI reference is an opaque pointer from C to a Java object. Local JNI references are limited to the current calling context and thread. Different threads and calling contexts may access global references. Weak global references are the same as global references, but do not prevent the garbage collector from collecting their referent. A JVM resource is a JVM-internal object used through JNI. In particular, a monitor is a mutual exclusion primitive, and strings and arrays are treated as resources if JNI pins or copies them. The programmer must manually call acquire and release functions for all JNI resources except for local references, which are managed semi-automatically.

APIs with manual or semi-automatic memory management suffer from well-known problems. Figure 7 shows the state of a single resource (in contrast to Figures 4 and 5, which show the state of an entire JVM thread). An acquire at insufficient capacity causes an overflow; a missing release at the end of the lifetime causes a leak; a use after releasing corrupts JVM state through a dangling reference; and a second release after releasing is a double-free. Jinn checks for all these errors.

For the five kinds of manually-managed resources shown in Figure 6, JNI has separate acquire and release APIs, and Jinn checks that they do not enter any of the error states from Figure 7. Jinn enters the resource into a table upon acquire, and removes it from that table upon release. This bookkeeping is practical when programs keep a modest number of manually-managed resources. If this assumption does not hold, the developer can configure Jinn to only check resource management during testing.

Jinn checks the four error cases from Figure 7 as follows. (1) The JVM throws an out-of-memory exception upon *overflow*; Section 4.1 describes how Jinn checks for exception states. (2) A *leak* becomes evident when the JVM terminates before the resource is released; Jinn detects that when Jinn's resource table is non-empty upon JVM shutdown. (3) Jinn can not always check for dangling references because it only interposes on language transitions; but it can check for one important case, namely when the user passes a dangling reference (such as a released global reference) to a JNI function, in which case Jinn reports an error. (4) Jinn reports a *double-free* if the user releases a resource that is no longer stored in Jinn's resource table.

Manually-managed resources: No overflow, leak, dangling reference, or double-free.
State: A list of allocated resources.
Check: Verify C code handles overflow and does not leak, use-after-release, and double-free resources.

JNI manages local references semi-automatically: acquire and release are more often implicit than explicit. Native code implicitly *acquires* a local reference when a Java native call passes it to C, or when a JNI function returns it. The JVM *releases* local references automatically when native code returns to Java, but the user can also manually release one (`DeleteLocalRef`) or several (`PopLocalFrame`) local references. Jinn enters the resource into a table upon acquire and removes it upon release.

Semi-automatic resources: No overflow, leak, dangling reference, or double-free.
State: Map lists of local references and capacities.
Check: Verify number of local reference is below capacity, and check leak, dangling-reference, and double-free.

Jinn performs bookkeeping to support overflow checks. The JNI specification only guarantees space for up to 16 local references. If more are needed, the user must explicitly request additional capacity with `PushLocalFrame`, and later release that space with `PopLocalFrame`. Jinn keeps track of local frames and checks the four error cases from Figure 7 as follows. (1) Jinn detects *overflow* if the current local frame has been used to capacity. (2) JNI releases individual local references automatically; Jinn checks for *leaked* local reference frames when native code returns to Java. (3) Jinn checks that local references passed as parameters to JNI functions are not *dangling* and, furthermore, belong to the current thread. (4) A *double-free* occurs when `DeleteLocalRef` is called twice for the same reference, or when there is nothing left to pop for a `PopLocalFrame`.

5. Experimental Results

This section evaluates the performance, coverage, and usability of Jinn to support our claim that it is the most practical FFI bug finder to date.

5.1 Methodology

Experimental environments. We used two production JVMs, Sun Hotspot Client 1.6.0_10 and IBM J9 1.6.0 SR5. We conducted all experiments on a Pentium D T3200 with 2GHz clock, 1MB L2 cache, and 2GB main memory. The machine runs Ubuntu 9.04 on Linux kernel 2.6.28-11.

JNI programs. We used several JNI programs: microbenchmarks, SPECjvm98, DaCapo, Subversion 39004 (2009-08-31), Java-gnome-4.0.10, and Eclipse 3.4. The microbenchmarks are a collection of eleven small synthetic JNI programs that violate constraints. Each microbenchmark violates one dynamic constraint and illustrates a programming pitfall [16] in the third column from Table 1. SPECjvm98 and DaCapo are written in Java, but exercise native code in the system library. Java-gnome-4.0.10 and Eclipse mix Java and C in user-level libraries. Except for Eclipse 3.4, we use fixed inputs.

Dynamic JNI checkers. We compare three dynamic JNI checkers: runtime checking in IBM and SUN JVMs, which is turned on by the `-Xcheck:jni` option, and Jinn, which is turned on by the `-agentlib:jinn` option in any JVM.

Experimental data. We collected timing and statistics results by taking the median of 30 trials to statistically tolerate experimental noise. The runtime systems show non-deterministic behavior from a variety of sources: micro-architectural events, OS scheduling, and adaptive JIT optimizations.

5.2 Performance

This section evaluates the performance of Jinn. Table 3 shows the results. Jinn adds instructions to every language transition between a JVM and native library to interpose and check transitions. The second column counts the total number of transitions between Java and C in the system library in the JVM. The third column is execution time of runtime checking for Hotspot 1.6.0_10. For Jinn, interposing transitions in the fourth column adds framework level overhead, and checking transitions in the fifth column represents the total overhead. Execution times are normalized to the production run of Hotspot 1.6.0_10 without runtime checking.

The transition count in the first column does not correspond directly to the overall execution overhead, because the transitions are a total number, whereas the overhead is normalized. For instance, *jython* has the highest total transition count, but not the highest relative overhead. The *fop* benchmark showed the highest overhead, which is consistent over both runtime checking and Jinn because transitions are a significant part of *fop*'s overall execution.

On average, Jinn has a modest 14% execution time overhead, which is 10% more overhead than for runtime checking. This result is quite natural, because the runtime checking inside the JVM does not have to pay the 8% overhead of interposing transitions. If we subtract the interposition overhead, Jinn's pure overhead is only 6%, which is comparable to the overhead of run time checking.

Benchmark	Environmental transition counts	Normalized execution time		
		Runtime checking	Jinn	
			Interposing transitions	Checking transitions
antlr	460,904	1.09	1.02	1.12
bloat	816,188	1.02	1.10	1.18
chart	981,481	1.03	1.08	1.16
eclipse	8,816,273	1.06	1.15	1.22
fop	1,977,263	1.10	1.12	1.47
hsqldb	205,191	1.03	1.09	1.12
jython	56,342,261	1.03	1.06	1.16
luindex	1,385,126	1.06	1.08	1.15
lusearch	3,369,800	0.99	1.05	1.15
pmd	905,431	1.03	1.07	1.12
xalan	1,153,419	1.04	1.12	1.15
compress	15,929	1.01	1.07	1.08
jess	152,986	1.05	1.21	1.14
raytrace	30,195	1.04	1.09	1.13
db	133,853	1.02	1.02	1.02
javac	255,019	1.00	1.02	1.02
mpegaudio	46,314	1.02	1.01	1.02
mtrt	31,864	0.99	1.13	1.14
jack	1,303,835	1.09	1.14	1.28
GeoMean		1.04	1.08	1.14

Table 3. Performance characteristics of Jinn with Hotspot VM 1.6.0_10.

```
WARNING in native method: JNI call made with exception pending at
  BadErrorChecking.call(Native Method) at
  BadErrorChecking.main(BadErrorChecking.java:5)
WARNING in native method: JNI call made with exception pending at
  BadErrorChecking.call(Native Method) at
  BadErrorChecking.main(BadErrorChecking.java:5)
```

(a) Hotspot

```
JVMJNCK028E JNI error in GetMethodID: This function
  cannot be called when an exception is pending
JVMJNCK077E Error detected in BadErrorChecking.call()V
JVMJNCK024E JNI error detected. Aborting.
JVMJNCK025I Use -Xcheck:jni:nonfatal to continue running
  when errors are detected.
Fatal error: JNI error
```

(b) J9

```
Exception in thread "main" JNIAssertionFailure:
 An exception is pending in CallVoidMethod.
  at jinn.JNIAssertionFailure.assertFail
  at BadErrorChecking.call(Native Method)
  at BadErrorChecking.main(BadErrorChecking.java:5)
Caused by: jinn.JNIAssertionFailure:
 An exception is pending in GetMethodID.
  ... 3 more
Caused by: java.lang.RuntimeException:
 checked by native code
  at BadErrorChecking.foo(BadErrorChecking.java:9)
  ... 2 more
```

(c) Jinn

Figure 8. Error messages from JVM runtime checkings, and Jinn for a microbenchmark violating the *exception state* constraint in Section 4.1.

5.3 Coverage of Jinn and JVM Runtime Checking

This section evaluates coverage of Jinn and shows that Jinn covers qualitatively and quantitatively more JNI bugs than the state-of-art dynamic checking in production JVMs. We run our eleven microbenchmark with Hotspot, J9, and Jinn.

Quality. Figure 8 compares representative error messages from Hotspot, J9, and Jinn for the *BadErrorChecking* microbenchmark, which violates the exception state constraints in Section 4.1. The C code in the benchmark ignores an exception from Java code, and calls two JNI functions: `GetMethodID` and `CallVoidMethod`. Hotspot reports that there were two illegal JNI calls, but does not identify the offending JNI function calls. J9 reports the first JNI function (`GetMethodID`), but does not show the calling context for the first bad JNI call because J9 aborts the JVM.

Jinn reports all the illegal JNI calls and their calling contexts, and includes the source location of the Java exception at line 9 of `BadErrorChecking.java`. In addition to the precise report, Jinn's error handler naturally inter-operates with debuggers. For instance, `jdb` and Eclipse `JDT` may catch the checker's custom exception, and programmers can inspect Java state to find the cause of failure. With Jinn, the Blink will present the programmer a full program state at the JNI error [14]. For instance, Blink presents the full calling context consisting of both Java and C frames.

Quantity. The behaviors of the production runs vary among running, crash, and null pointer exception, and none of them is correct. The runtime checking in the JVMs shows inconsistent behavior in more than half of the microbenchmark (7 of 11). Jinn is the only dynamic bug-finder that consistently covers the JNI bugs in the eleven microbenchmark and throws an exception. Quantitative coverages of Jinn, Hotspot, and J9 are 100%, 63% (7/11), and 63% (7/11) when bug reports are warning, error, and exception. Jinn's bug reports were 21% larger than the combined reports from Hotspot and J9 (9/11).

5.4 Usability with Open Source Programs

This section evaluates the usability of Jinn based on our experience of running Jinn over Subversion, Java-gnome, and Eclipse. All these open-source programs are in wide industrial and academic use with a long revision history. These case studies show how Jinn finds errors in widely-used programs.

5.4.1 Subversion

Running Subversion's regression test suite under Jinn, we found two overflows of local references and one dangling local reference.

Overflow of local reference. Jinn found that Subversion allocated more than 16 local reference in two call sites to JNI functions: line 99 in `Outputer.cpp` and line 144 in `InfoCallback.cpp`. The original program with the leak allocated more than 16 local references without asking for more capacity. With Jinn, when the number of active local references reaches the capacity (e.g., 16) and the program calls a JNI function, Jinn throws an exception. The problematic allocation source line is:

```
jstring jreportUUID = JNIUtil::makeJString(info->repos_UUID);
```

After looking at the calling context, we found that the program misses a call to `DeleteLocalRef`. We inserted the following lines:

```
env->DeleteLocalRef(jreportUUID);
if (JNIUtil::isJavaExceptionThrown()) return NULL;
```

After re-compiling, the program passed the regression test. With our patch, the number of active local references never exceeded 8.

Use of dangling local reference. The use of a dangling local reference happens at the execution of a C++ destructor when the C++ variable `path` goes out of scope in file `CopySources.cpp`.

```
{
   JNIStringHolder path(jpath);
   env->DeleteLocalRef(jpath);
} /* The destructor of JNIStringHolder is executed here. */
```

At the declaration of `path`, the constructor of `JNIStringHolder` stores the JNI local reference `jpath` in the member variable `path::m_jtext`. Later the call `DeleteLocalRef` releases the `jpath` local reference, and thus, `path::m_jtext` becomes dead. When the program exits from the C++ block, it calls the destructor of `JNIStringHolder`. Unfortunately, this destructor uses the dead JNI local reference:

```
JNIStringHolder::~JNIStringHolder() {
  if (m_jtext && m_str)
    m_env->ReleaseStringUTFChars(m_jtext, m_str);
}
```

The JNI function `ReleaseStringUTFChars` uses the dangling JNI reference (`m_jtext`). This bug is not syntactically visible to the programmer because the C++ destructor feature obscures control flow when releasing resources. In our experience, this bug did not directly crash production JVMs. We looked at the internal implementation of `ReleaseStringUTFChars` in an open-source Java virtual machine (Jikes RVM). In Jikes RVM, `ReleaseStringUTFChars` ignores its first parameter, and thus, it does not matter that it is a dangling reference. If other JVMs are implemented similarly, this bug will remain hidden. But the code represents a time bomb, because the bug will be exposed as soon as the program runs on a JVM where the implementation of `ReleaseStringUTFChars` uses its first parameter. For example, a JVM may represent strings in UTF8 representation internally as proposed by Zilles [24] and then share them directly with JNI.

5.4.2 Java-gnome

Running Java-gnome's regression test suite under Jinn, we found one nullness bug and one dangling local reference.

Nullness. We rediscovered a bug that was previously reported by Lee et al. in their Blink debugger paper [14]. Note, however, that unlike their work, Jinn does not require debugger.

Use of dangling local reference. We rediscovered a bug that was reported to the Java-gnome developers as bug 576111. A Java-gnome developer confirmed that the bug should be fixed. It violates a constraint of semi-automatic resources. Jinn reports that Line 348 of `binding_java_signal.c` violates a constraint:

```
(*env)->CallStaticVoidMethodA(env, bjc->receiver,
                              bjc->method, jargs);
```

The `bjc->receiver` is a dead local reference. This bug did not crash Hotspot and J9, but as noted before, bugs that are only benign due to implementation characteristics of a specific JVM vendor and are represent time bombs and should be fixed.

5.4.3 Eclipse 3.4

We opened a Java project in Eclipse-3.4, and Jinn reported one violation of the entity-specific subtyping constraint in line 698 of `callback.c` in its SWT 3.4 component.

```
result = (*env)->CallStaticSWT_PTRMethodV(env, object, mid, vl);
```

The `object` must point to a Java class which has a static Java method identified by `mid`. The actual class did not have the static method, but its superclass declares the method. It is challenging for the programmer to ensure this constraint, because the source of the error involves dynamic call-back control and a Java inner class. Because the production JVM may not use `object`, this bug survived multiple revisions.

6. Generalization

To demonstrate that our approach generalizes to other languages, we apply it to the Python/C API [1]. This FFI is similar to JNI in that its FFI usage rules reduce into the same three classes. The Python/C API differs from JNI in that it is lower-level and bakes in some of the Python interpreter's implementation details. This tight coupling between interface and implementation makes it more difficult for us to interpose upon language transitions. This section describes how the usage rules for the Python/C API resemble and depart from JNI rules. We synthesized a dynamic checker for a subset of these rules.

6.1 Python/C Constraint Classification

Like JNI, the Python/C API specification describes numerous rules which constrain how programmers can combine Python and C. These constraints fall into the same basic classes we discussed in Section 4: (1) interpreter state constraints, (2) type constraints, and (3) resource constraints.

State constraints. The Python/C API constrains the behavior of exceptions and threads. Python/C's exception constraints are similar to JNI's. Python/C's thread constraints differ from JNI's, because Python's threading model is simpler than Java's. For each instantiation of the Python interpreter, a thread must possess the Global Interpreter Lock (GIL) to execute. The Python interpreter contains a scheduler that periodically acquires and releases the GIL on behalf of a program's threads.

The Python/C API permits C code to release and re-acquire the GIL around blocking I/O operations. The API permits C code to create its own threads and bootstrap them into Python. Because C code may manipulate thread state directly, the programmer may write code that deadlocks. For example, the programmer may accidentally acquire the GIL twice. The book-keeping mechanisms for JNI from Section 4.1 also apply to Python/C.

Type constraints. Because Python is a dynamically typed language, Python/C's types are less constrained than JNI's. The Python interpreter performs dynamic type checking for many operations on built-in types. Sometimes, however, the interpreter forgoes these checks—as well as some null-checks—for performance reasons. If a program passes an unexpected value to a Python/C API, the program may crash or exhibit undefined behavior. Interposing on these calls would slow the interpreter, but would enable more reliable bug-reporting for mixed-language programs.

Resource constraints. Python employs reference counting for memory management. In Python code, reference counting is transparent and fully automatic. Native code programmers must instead manually increment and decrement a Python object's reference count, according to the API manual's instructions. The Python/C API manual defines a notion of *reference ownership*. Each reference that co-owns an object is responsible for decrementing the object's reference count when it no longer requires that object. Neglecting decrements leads to memory leaks. C code may keep and *borrow* a reference. Borrowing a reference does not increase its reference count, but using a borrowed reference to a freed object causes a dangling reference error. The Python/C API manual describes co-owned and borrowed references, and it specifies which kinds of references are returned by the various API functions. A dynamic checker must track the state of these references in order to report usage violations to the user.

Figure 9 contains an example Python/C function that mismanages its references. The reference `first` in Line 6 is borrowed from the reference `pythons`. When Line 8 decrements the reference count for `pythons`, it dies. The Python/C API manual states that the program should no longer use `first`, but the program uses this reference at Line 10. This use is a dangling reference error, and leads to inconsistent program behavior.

Although the usage classes for Python/C are similar to those of JNI, the design and implementation of Python/C presents some challenges to synthesizing a dynamic checker for Python that interposes on language transitions.

```
1.  static PyObject* dangle_bug(PyObject* self, PyObject* args) {
2.    PyObject *pythons, *first;
3.                /* Create and delete a list with a string element.*/
4.    pythons = Py_BuildValue("[ssssss]",
5.       "Eric", "Graham", "John", "Michael", "Terry", "Terry");
6.    first = PyList_GetItem(pythons, 0);
7.    printf("1. first = %s.\n", PyString_AsString(first));
8.    Py_DECREF(pythons);
9.                                        /* Use dangling reference. */
10.   printf("2. first = %s.\n", PyString_AsString(first));
11.              /* Return ownership of the Python None object. */
12.   Py_INCREF(Py_None);
13.   return Py_None;
14. }
```

Figure 9. Python/C API dangling reference error. The borrowed reference first becomes a dangling reference when pythons dies.

6.2 Interposing Transitions

Our dynamic checking is based on language interposition. In the case of JNI, the JVMTI provides interposition. Unfortunately, the Python interpreter does not have JVMTI-equivalent interface. We analyzed the `Python.h` header file for Python 2.6.4 and modified the interpreter to implement interposition. In the course of our modification, we faced and solved three challenges: (1) variadic functions with missing non-variadic counterparts, (2) prevalent use of C macros, and (3) use of Python/C functions inside the interpreter.

Variadic functions. A variadic C function takes a variable number of arguments. Wrapping variadic functions requires an equivalent non-variadic function. For instance, `printf` takes a variable number of arguments, and the wrapper may use its equivalent non-variadic function, `vprintf`. Seventeen functions in Python/C are variadic, but twelve of them do not have a non-variadic equivalent. We added twelve non-variadic functions to the Python interpreter, so that the dynamic analysis may wrap all of Python/C's variadic functions. For instance, we added a `PyObject_VaCallFunction` for the variadic function `PyObject_CallFunction`. This extension was almost mechanical because the variadic functions internally contain code for non-variadic functions.

Macro functions. The Python/C API makes extensive use of C macros. Some macros directly modify interpreter state without executing a function call. Because the Python/C API does not contain a function call for this behavior, our dynamic analysis has nothing to interpose and it cannot track the behavior. We replaced five macros that manage cross-lingual references with equivalent functions, so that our analysis may interpose upon reference management. The remaining approximately 300 macros needed no replacement, because they already correspond to a Python/C API function or they do not modify interpreter state. Programmers must re-compile their extension modules against our customized interpreter, but they do not need to change their code.

Shared use of Python/C. Both the Python interpreter and user modules can call Python/C API functions, which makes it difficult to detect application language transitions. Furthermore, performing dynamic analysis during interpreter-internal calls would significantly slow down the interpreter. To solve this problem, we duplicate approximately 400 Python/C functions into two versions, one public and one interpreter-internal. We then used automatic code-rewriting to change the interpreter so it only calls the internal versions, which do not perform interposition. The public versions wrap the internal versions and perform interposition.

6.3 Synthesizing Dynamic Checkers

We implemented an interposition framework for Python/C and used it to synthesize a use-after-release checker. The synthesizer takes a specification file that lists which functions return new or borrowed references. The resulting synthesized checker can detect the dangling reference bug in Figure 9. The checker keeps track of reference state for borrowed and co-owned references. Each borrower is associated with a co-owner. For example, the checker determines that `python` is a co-owned reference and that `first` borrows from `python`. When a co-owner relinquishes its hold on a reference by decrementing its count, all its borrowed references become invalid. If the program references invalid, borrowed references—as Figure 9 does at Line 10—then the checker signals an error.

Our checker permits each borrowed reference to correspond to only one co-owner. The Python/C API manual is not clear on whether a borrowed reference can correspond to multiple co-owners. If a borrowed reference may have multiple co-owners, then the checker may report false positives. We could enforce a many-owner model for borrowed references.

7. Conclusion

Many programs are multi-lingual, and bugs at the language boundary are frequent. We propose specifying foreign function interfaces (FFIs) with state machines, which encode how to properly deal with cross-language state, type, and resource constraints. This paper shows how to use synthesis to automate turning these state machine specifications into dynamic bug checkers. We applied our approach to synthesize dynamic bug checkers for two FFIs: the FFI between Java and C, and the FFI between Python and C. Our synthesized bug checker for the Java/C FFI is called Jinn, and has uncovered previously unknown bugs in widely-used Java native libraries. Our approach to multi-lingual bug correction is the most practical and effective one to date.

Acknowledgments

We thank Mike Bond, Jungwoo Ha, and Jennifer Sartor for helpful discussions; Mike Bond for helpful feedback on the text and his suggestion of wrapping Python/C macros; Jungwoo Ha for explaining some details of hardware performance counter libraries; and Jennifer Sartor for referring to Accordion Arrays.

References

[1] Python/C API reference manual. `http://docs.python.org/c-api`, Nov. 2009.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2005.

[3] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2008.

[4] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *USENIX Tcl/Tk Workshop (TCLTK)*, 1996.

[5] F. Chen and G. Rosu. MOP: An efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[6] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering (ICSE)*, 1999.

[7] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *Programming Language Design and Implementation (PLDI)*, 2005.

[8] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *European Symposium on Programming (ESOP)*, 2006.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, June 2005.

[10] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Programming Language Design and Implementation (PLDI)*, 2002.

[11] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[12] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, Apr. 1988.

[13] G. Kondoh and T. Onodera. Finding bugs in Java native interface programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[14] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley. Debug all your code: Portable mixed-environment debugging. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[15] S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *Computer and Communications Security (CCS)*, 2009.

[16] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[17] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Principles of Programming Languages (POPL)*, 2002.

[18] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit. Automatic generation of library bindings using static analysis. In *Programming Language Design and Implementation (PLDI)*, 2009.

[19] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, Feb. 2000.

[20] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe Java native interface. In *International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[21] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In *Usenix Security Symposium (SS)*, 2008.

[22] G. Tan and G. Morrisett. ILEA: Inter-language analysis across Java and C. In *Object-Oriented Programming Systems and Applications (OOPSLA)*, 2007.

[23] The GNOME Project. GNOME bug tracking system. Bug 576111 was opened 2009-03-20. `http://bugzilla.gnome.org`.

[24] C. Zilles. Accordion arrays: Selective compression of unicode arrays in Java. In *International Symposium on Memory Management (ISMM)*, 2007.