

Jinn: Synthesizing Dynamic Bug Detectors for Foreign Language Interfaces

Byeongcheol Lee[†] Ben Wiedermann[†] Martin Hirzel[‡] Robert Grimm[§] Kathryn S. McKinley[†]

[†]University of Texas at Austin
{bcee,ben,mckinley}@cs.utexas.edu

[‡]IBM Watson Research Center
hirzel@us.ibm.com

[§]New York University
rg Grimm@cs.nyu.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 is not, and this poor specification is a source of many errors in *multilingual* programs. For example, virtually all Java programs compose Java and C using the Java Native Interface (JNI). Since JNI is informally specified, developers have difficulty using it correctly, and current Java compilers and virtual machines (VMs) inconsistently check only a subset of JNI 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. We identify three classes of FFI constraints encoded by eleven state machines that capture thousands of JNI and Python/C FFI rules. We use a mapping function to specify which state machines, transitions, and program entities (threads, objects, references) to check at each FFI call and return. From this function, we synthesize a context-specific dynamic analysis to find FFI bugs. We build bug detection tools for JNI and Python/C using this approach. For JNI, we dynamically and transparently interpose the analysis on Java and C language transitions through the JVM tools interface. The resulting tool, called Jinn, is compiler and virtual machine *independent*. It detects and diagnoses a wide variety of FFI bugs that other tools miss. This approach greatly reduces the annotation burden by exploiting common FFI constraints: whereas the generated Jinn code is 22,000+ lines, we wrote only 1,400 lines of state machine and mapping code. Overall, this paper lays the foundation for a more principled approach to developing correct multilingual software and a more concise and automated approach to FFI specification.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification—Assertion Checkers, Reliability; D.2.5 [Software Engineering]: Testing and Debugging—Debugging Aids; D.3.4 [Programming Languages]: Processors—Run-time Environments

General Terms Design, Languages, Reliability, Verification

Keywords Multilingual Programs, Foreign Function Interfaces (FFI), Java Native Interface (JNI), Python/C, Dynamic Analysis, FFI Bugs, Specification, Specification Generation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019/10/06...\$10.00

1. Introduction

Programming language designers spend years precisely specifying languages [12, 15, 24]. Likewise, language implementors exert considerable effort enforcing specifications with static and dynamic checks. Many developers, however, compose languages to reuse legacy code and leverage the languages best suited to their needs. Such multilingual programs require additional specification due to syntactic and semantic language differences. Well-designed and well-specified *foreign function interfaces* (FFIs) include recent integrated language designs [13, 25] and language binding synthesizers [5, 22], but programmers have not yet widely adopted them.

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 and complex usage rules. For example, JNI has 229 API calls and 1,500+ rules, although it is well-encapsulated and portable [19]. This complexity arises due to language impedance mismatches in object models, type systems, memory management, and exceptions.

Example JNI calls include: look up a class by name, specified as a string, and return its class descriptor; look up a method by class descriptor and signature, again specified as a string, and return a method descriptor; and invoke a method by its descriptor. Example JNI constraints include: when a program calls from C to Java, the actuals must match the formals' declared types; also when calling from C to Java, a program must not have a pending Java exception; and a program must issue an extra request to maintain more than 16 local cross-language references.

While language differences easily lead to large and complex FFIs, they also make complete static enforcement of FFI usage rules impossible. For example, JNI's use of string arguments to describe class names and method signatures cleanly abstracts over the underlying JVM and its implementation. Unfortunately, these strings prevent standard static type checking, and even advanced interprocedural static analysis is incomplete [10, 28]. Similarly, complete static checking of pending exceptions and the count of local references is impossible.

Overall, voluminous, complex, and context-dependent FFI rules lead to three problems. (1) Some rules are statically undecidable and require dynamic validation. (2) Language implementations are inconsistent and often diverge when they do dynamically enforce a rule, for example, by signaling an exception versus terminating a program. (3) Writing correct FFI code that follows all rules is hard for programmers. As a direct consequence of these three problems, real-world multilingual programs are full of FFI bugs [10, 11, 16, 17, 18, 26, 27].

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 three classes of rules:

JNI Pitfall	Default Behavior		Language Design	Static Analysis	Dynamic Analysis		
	HotSpot	J9			HotSpot	J9	Jinn
1. Error checking	running	crash	[13], [25]	[16], [18]	warning	error	exception
2. Invalid arguments to JNI functions	running	crash	[13], [25]	[11], [27]	running	crash	exception
3. Confusing jclass with jobject	crash	crash	[13], [25]	[11]	error	error	exception
6. Confusing IDs with references	crash	crash	[13]	[11]	error	error	exception
8. Terminating Unicode strings	running	NPE	[13], [25]		running	NPE	running/NPE
9. Violating access control rules	NPE	NPE	[13], [25]		NPE	NPE	exception
11. Retaining virtual machine resources	leak	leak	[13], [25]	[16]	running	warning	exception
12. Excessive local reference creation	leak	leak			running	warning	exception
13. Using invalid local references	crash	crash	[13]	[16]	error	error	exception
14. Using the JNIEnv across threads	running	crash	[13]		error	crash	exception
16. Bad critical region	deadlock	deadlock	[13]	[16]	warning	error	exception

Table 1. JNI pitfalls [19]. *Running*: continues to execute in spite of undefined JVM state. *Crash*: aborts without diagnosis. *Warning*: prints diagnosis and continues. *Error*: prints diagnosis and aborts. *NPE*: raises a null pointer exception. *Exception*: raises a Jinn JNI exception.

JVM state constraints restrict JVM thread context, critical section state, and/or exception state.

Type constraints restrict parameter types, values (e.g., not NULL), and semantics (e.g., no writing to final fields).

Resource constraints restrict the number of multilingual pointers and resource lifetimes, e.g., locks and memory.

Furthermore, eleven state machines express all these rules. Compared to Liang’s specification, this concision is unexpected [19].

Using the state machines, we present a new way to synthesize context-sensitive dynamic analysis that precisely identifies FFI violations. We parameterize each state machine by program *entities*: threads, references, and objects. The synthesizer attaches state machines to JNI parameters, return values, and threads. For JNI analysis, we observe it is sufficient to transition state machines and check constraints at language boundaries. Consequently, the synthesizer inserts wrapper functions with generated code before and after FFI calls. These functions transition the state machines and dynamically verify the FFI specification. Whereas we wrote about 1,400 non-comment lines of state machine and mapping code for JNI, our tool generates 22,000+ lines, demonstrating that this approach greatly reduces the annotation burden for FFI specification.

We dynamically and transparently inject the analysis into user code using language interposition. We implement interposition with vendor-neutral JVM interfaces in a tool called Jinn. 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 JNI constraints at every language transition and diagnoses bugs when the program violates a constraint.

Jinn is more practical than the state-of-the-art for finding JNI bugs, in part because it does not depend on the JVM or C compiler. 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. Table 1 compares Jinn to prior work qualitatively and quantitatively. It shows that prior dynamic tools are incomplete. For the subset of errors that static analysis finds, they are preferable since Jinn only finds exercised bugs. However, static tools cannot conclusively identify many dynamic resource bugs and type errors that Jinn finds with no false positives.

We explore the generality of our FFI specification and approach using the Python/C FFI. We find that Python/C requires the same three rule classes and similar state machines. Just as for Java and C, the Python/C rules reflect fundamental language semantic mismatches. Whereas C relies on manual resource management and is weakly typed, Python automates memory reclamation, provides

a stronger typing discipline (in this case dynamic), enforces well-formed resource access, and automates error propagation. We implement and evaluate a Python/C bug detector for a few unique rules, providing further evidence for the generality of our approach.

In summary, the contributions of this paper include:

- Synthesis of dynamic FFI analysis from state machines.
- The first rigorous state machine specification of JNI based on its informal specification [19].
- Jinn, the most practical JNI bug detector, which is generated by our synthesizer.
- 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 specifying the three classes of FFI constraints, this paper helps lay a foundation for a more principled approach to developing correct multilingual software, foreign function interfaces, and perhaps other interfaces.

2. Motivation and Related Work

This section shows that JNI has inconsistent implementations, which likely stems from poor specification and certainly complicates portable JNI programming. It then quantitatively and qualitatively compares Jinn to prior work that uses language design, static analysis, and dynamic analysis to diagnose FFI bugs. Finally, it reviews other work on synthesizing analyses from state machines.

FFI programming is challenging because programmers must reason about multiple languages and their semantic interactions. For example, Chapter 10 of the JNI manual identifies fifteen pitfalls [19]. We list the most serious of these in Table 1, using Liang’s numbering scheme, and include “bad critical region” from Chapter 3.2.5 as a 16th pitfall. We created small JNI programs to exercise each pitfall and executed them with HotSpot and J9. Columns two and three show that JNI mistakes cause a wide variety of crashes and silent corruption. The two JVMs behave differently on four of the pitfalls. Columns six and seven show the JVMs are not much better with built-in JNI checking (turned on by 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 [11, 13, 16, 18, 25, 27]. We did not execute the static tools. Language designs cover the widest class of JNI bugs [13, 25], but

new languages require developers to rewrite their code. Static analysis catches some, but not all, pitfalls. For example, statically enforcing non-nullness without language support (e.g., a `@NonNull` annotation) is undecidable. At the same time, dynamic and static FFI analysis are complementary. Dynamic analysis misses unexercised bugs, whereas static analysis reports false positives.

The last column shows that Jinn detects all but one of these serious and common errors. Pitfall 8 depends on how C code uses character buffers and requires analysis or instrumentation of a program's entire C code, which is beyond our more targeted dynamic analysis. Consequently, the program exhibits the same behavior as a production run without Jinn, i.e., it either keeps on running (HotSpot) or signals a null pointer exception (J9). 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 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 [25] combines Java with CCured [21], and Jeannie safely and directly nests Java and C code into each other using quasi-quoting [13]. 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 economical than spending time to fix them after the fact. Another approach generates language bindings for annotated C and C++ header files [5, 14]. Ravitch et al. reduce the annotations required for generating idiomatic bindings [22]. Jinn is more practical than these approaches, because it does not require developers to rewrite or annotate their code in a different language.

2.2 Static FFI Bug Checkers

A variety of static analyses verify foreign function interfaces [10, 11, 16, 18, 26, 27]. All static FFI analysis approaches suffer from false positives because the specification includes dynamic properties, such as non-null reference parameters, valid Java class and method names in string parameters, and less than 16 local references. Static analysis cannot typically guarantee these properties. For instance, J-Saffire reports false positives and warnings [11]; Tan et al. report a false positive rate of 15.4% [18]; and BEAM reports a false positive, while missing the bug in Section 3.1. In contrast, Jinn never generates false positives, but only finds bugs actually triggered during program execution. Furthermore, whereas prior static analyses for JNI require the native library to be written in C and available in source form, Jinn is neither restricted to C nor does it require source code access. For instance, Jinn found FFI bugs in the Subversion Java binding written in C++. In summary, static analysis finds a subset of FFI bugs without executing the program, but suffers from false positives. In comparison, Jinn finds more FFI bugs, but only when they are exercised; suffers from no false positives; and requires no source code access.

2.3 Dynamic FFI Bug Checkers

Some JVMs provide built-in dynamic JNI bug checkers, enabled by the `-Xcheck:jni` command-line flag. While convenient, these error checkers only cover limited classes of bugs, and JVMs implement them inconsistently. NaturalBridge's BulletTrain ahead-of-time Java compiler performed several ad-hoc JNI integrity checks on language transitions [20]. The Blink debugger provides JNI bug checkers that work consistently for different JVMs, but its coverage is limited to two bugs: validating exception state and nullness con-

straints [17]. These 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 JVM Tools Interface (JVMTI), and explicitly throws an exception at the point of failure. Exceptions provide a principled and language supported approach to software quality, for example, enabling a GUI-based program to report the bug in a dialog instead of relying on the user to sift through the system log. Furthermore, when the exception's error message and calling context do not suffice to identify the cause of the failure, programmers can rerun the program with both Jinn and a Java debugger. The debugger then catches the exception, and the programmer can access the detailed program state at the point of failure.

2.4 State Machine Specifications

Several programmable bug checkers take state machine specifications, and report errors when state machines reach error states. For instance, Metal [9] and SLIC [4] are languages for specifying state machines that are then used to find bugs through static analysis. Dwyer et al. survey state-machine driven static analyses [8]. On the dynamic side, 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 [7]; and Arnold et al. implement FSMs for bug detection in a JVM, controlling the runtime overhead by sampling [3]. While in principle these specification languages are expressive enough to describe many FFI constraints, in practice none of them address the unique challenges of multi-lingual software. Also, unlike Jinn, most of them require source code access.

3. An Example JNI Bug and Detector

This section illustrates and motivates our approach using an example. It describes some JNI background, an example JNI bug, and a state machine that captures this bug. It then describes how to use this state machine to dynamically detect the example bug on language transition boundaries at JNI calls and returns. We explain how our system attaches state machines to program *entities*, i.e., objects, individual references, and threads, and how it transitions entity states on JNI calls and returns, i.e., through explicitly passed arguments and results as well as the implicit threads.

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. However, 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 the garbage collector to pin arrays. Developers using JNI avoid indirection overhead on the C side by, for example, caching method and field IDs, and pinning resources. At the same time, JVM developers avoid implementation complexity by requiring explicit calls to mark references as global and to release pinned objects.

3.1 Example FFI Bug

Figure 1 shows a simplified version of an FFI bug from the GNOME project's Bugzilla database (Bug 576111) [29]. GNOME is a graphical user interface that makes heavy use of several C libraries. In the example, Line 1 defines a C function `Java_Call`

```

1. JNIEXPORT void JNICALL Java_Callback_bind(JNIEnv *env,
2. jclass clazz, jclass receiver, jstring name, jstring desc)
3. {
4.     /* Register an event call-back to a Java listener. */
5.     EventCallBack* cb = create_event_callback();
6.     cb->handler = callback;
7.     cb->receiver = receiver; /* receiver is a local reference.*/
8.     cb->mid = find_java_method(env, receiver, name, desc);
9.     if (cb->mid != NULL) register_callback(cb);
10.    else destroy_callback(cb);
11. } /* receiver is a dead reference. */
12. static void callback(EventCallBack* cb, Event* event) {
13.     JNIEnv* env = find_env_pointer_from_current_thread();
14.     jvalue* jargs = marshal_event(cb, env, event);
15.     /* BUG: dereference of now invalid cb->receiver. */
16.     (*env)->CallStaticVoidMethodA(
17.         env, cb->receiver, cb->mid, jargs);
18. }

```

Figure 1. JNI invalid local reference error in a call-back routine from GNOME (Bug 576111) [29].

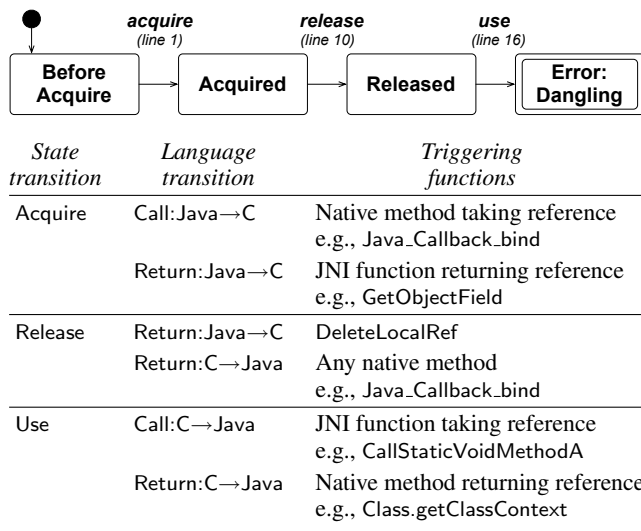


Figure 2. A resource tracking state machine for local references and the mapping from state transitions to Java/C language transitions (calls and returns) to dynamically detect the bug in Figure 1.

back.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 registers a new C heap object cb storing the receiver class and method name passed as parameters from Java. The C function callback referenced on Line 5 is defined starting at Line 11. It uses the cb parameter 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, which resides 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, the garbage collector would need to communicate with the C runtime about live references. Thus, cb->receiver becomes invalid when the function returns at Line 10. But Line 6 stores receiver in a heap object, letting it escape. When Line 16 retrieves receiver from the heap and uses it as a parameter to CallStaticVoidMethodA, it is an invalid dangling

```

1. void wrapped_Java_Callback_bind(JNIEnv *env,
2. jclass clazz, jclass receiver, jstring name, jstring desc)
3. {
4.     /* Instrument Call:Java→C for Acquire state transition. */
5.     jobject_set refs = jinn_acquire_thread_local_jobject_set();
6.     if (clazz != NULL) { jinn_refs_acquire(refs, clazz); }
7.     if (receiver != NULL) { jinn_refs_acquire(refs, receiver); }
8.     if (name != NULL) { jinn_refs_acquire(refs, name); }
9.     if (desc != NULL) { jinn_refs_acquire(refs, desc); }
10.    /* Call the wrapped native method. */
11.    Java_Callback_bind(env, clazz, receiver, name, desc);
12.    /* Instr. Return:C→Java for Release state transition. */
13.    jinn_release_thread_local_jobject_set(refs);
14. }

```

Figure 3. Wrapper for function Java_Callback_bind from Figure 1 with instrumentation for Acquire and Release state transitions.

```

1. void wrapped_CallStaticVoidMethodA(JNIEnv *env,
2. jclass clazz, jmethodID mid, jvalue *args)
3. {
4.     /* Instrument Call:C→Java for Use state transition. */
5.     jobject_set refs = jinn_get_thread_local_jobject_set();
6.     if ((clazz != NULL) && !jinn_refs_contains(refs, clazz)) {
7.         /* Raise a JNI exception. */
8.         return jinn_throw_JNIException(env, "Error: dangling");
9.     }
10.    /* Call the wrapped JNI function. */
11.    CallStaticVoidMethodA(env, clazz, mid, args);
12. }

```

Figure 4. Wrapper for function CallStaticVoidMethodA from Figure 1 Line 15 with instrumentation for Use state transition.

reference, and the JVM’s garbage collector may have either moved the object or reclaimed it and reused the corresponding memory.

The JNI specification merely says that this reference is invalid and leaves the consequences up to the vendor’s Java implementation [19]. 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 [16] misses this bug. The other static analyses in Table 1 can generate warnings on all such stores (e.g., at Line 6), but will also generate thousands of false alarms.

3.2 Example FFI Bug Detector

This section shows how to identify this bug dynamically using a state machine. Figure 2 shows a simplified state machine that enforces local usage rules, applied to the receiver parameter at runtime. On entry to the method (Figure 1: Line 1), the state of receiver transitions from Before Acquire to Acquired. When the method returns back to Java (Line 10), the state transitions from Acquired to Released. Finally, the call from C to Java at Line 16 uses the reference cb->receiver, triggering a transition to the Error: Dangling state and thus detecting the bug.

The table in Figure 2 shows more generally where state transitions occur. For example, dynamic analysis must execute the Acquire transition for all reference parameters on all calls from Java to C. On return from C to Java, dynamic analysis must execute the Release transition for all local references. To instrument both calls and returns, we wrap these calls. For example, our dynamic checker replaces Java_Callback_bind with the wrapper function wrapped_Java_Callback_bind shown in Figure 3. The instrumentation attaches state machines to entities (threads, parameters, and return values) by using thread-local storage (refs).

We also instrument the JNI functions that implement the C API for interacting with the Java virtual machine. For example, the Use transition in the table happens on calls from C to Java if the callee is a JNI function taking a reference, such as `CallStaticVoidMethodA`. Such a use is an error if the reference is in the Released state. Figure 4 shows the wrapper with the instrumentation.

For illustration purposes, these example wrappers omit other checks our system performs. For example, JNI limits the number of available local references, so there is another possible error state for overflow. Developers may manually manage the number of available local references with the JNI functions `PushLocalFrame` and `PopLocalFrame` and the corresponding dynamic analysis requires instrumentation to count references. The figures also omit checks for thread state, exception state, and parameter nullness. Section 5 explains all the constraints we check and their encoding in state machines.

4. Dynamic Analysis Synthesis

We use state machine specifications like the one in Figure 2 to synthesize a dynamic analysis. Each state machine specification describes state transitions, which are triggered by language transitions. Their cross-product yields thousands of checks in the dynamic analysis. For example, before executing the JNI call in Line 15 of Figure 1, the analysis enforces 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 did not disable GC to directly access Java objects including arrays.
- `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 specify state machines as follows.

Defining state machine states and transitions: Each FFI constraint is defined by a state machine. The individual states are encoded as C data structures and the transitions as C code, which also checks whether a transition has, in fact, been triggered. For example, the if-statement in Line 6 of Figure 4 is a transition check for determining whether the entity is currently in the Released state and should therefore transition to the Error: Dangling state. Each state machine specification M_i has a set of state transitions $M_i.stateTransitions$.

Mapping state transitions to language transitions: Each specification has a function $M_i.languageTransitionsFor$ that maps state transitions to language transitions. The synthesizer consults this mapping to inject context-specific instrumentation into wrapper functions. For example, Figure 2 illustrates a mapping. Figures 3 and 4 show generated wrappers. Each state transition $s_a \rightarrow s_b$ may occur at a set

$$L = M_i.languageTransitionsFor(s_a \rightarrow s_b)$$

of language transitions. Each language transition ℓ in this set is a record containing the fields `function`, `direction` (Call or Return), and `entities` (threads, parameters, and return values).

Applying state machines to entities: At runtime, the wrappers attach state machines to entities and then transition the entity-specific state machine(s) based on context, encoding the state machine states in thread-local storage. For example, the wrapper in Figure 3 associates a state machine with the receiver ref-

Algorithm 1 Input: state machine specifications M_1, \dots, M_n . Output: FFI wrapper functions instrumented with dynamic checker.

```

1: for each state machine specification  $M_i \in \{M_1, \dots, M_n\}$  do
2:   for each state transition  $s_a \rightarrow s_b \in M_i.stateTransitions$  do
3:     let  $L = M_i.languageTransitionsFor(s_a \rightarrow s_b)$ 
4:     for each language transition  $\ell \in L$  do
5:       let  $w$  be the wrapper for  $\ell$ .function
6:       add the following synthesized code to the start or
       end of  $w$ , depending on whether  $\ell.direction$  is Call or
       Return:
7:       for each entity  $e \in \ell.entities$  do
8:         if  $e$  satisfies the transition check for  $s_a \rightarrow s_b$  then
9:           modify the state machine encoding to record the
           transition of  $e$  from  $s_a$  to  $s_b$ .

```

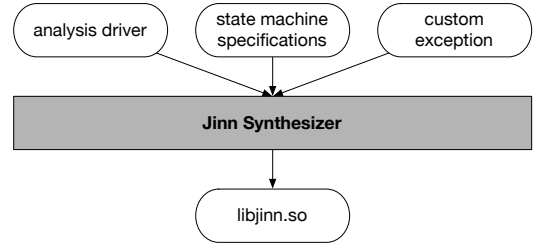


Figure 5. Structure of Jinn Synthesis.

erence, transitions its state to Acquired, and encodes this information by adding the reference to the thread-local list `refs`. As already mentioned above, the analysis developer specifies state machine encodings as a set of mutable data structures and functions that manipulate those structures.

The state machine specifications consisting of these three components (state transitions, mappings from state transitions to language transitions, and state machine encodings) serve as input to Algorithm 1. The algorithm computes the cross product of state transitions and FFI functions, and 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 5.

The synthesizer takes two additional inputs: an analysis driver and a custom exception. The output of the synthesizer is Jinn—a shared object file that the JVM dynamically loads using the JVM tools interface (JVMTI). The analysis driver initializes the state machine encodings and dynamically injects the generated, wrapped FFI functions into a running program. The custom exception defines how the dynamic analysis reports errors. 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.

We now turn our attention to specifying the state machines.

5. JNI Constraint Classification

This section describes how three classes of constraints summarize all JNI constraints and how to encode these constraints in eleven state machines. These JNI constraints encompass 1,500+ rules from the JNI manual. As far as we are aware, no other work specifies the JNI FFI, other FFIs, or any other API so concisely.

Constraint	Count	Description
<i>JVM state constraints</i>		
JNIEnv* state	229	Current thread matches JNIEnv* thread
Exception state	209	No exception pending for sensitive call
Critical-section state	225	No critical section
<i>Type constraints</i>		
Fixed typing	157	Parameter matches API function signature
Entity-specific typing	131	Parameter matches Java entity signature
Access control	18	Written field is non-final
Nullness	416	Parameter is not null
<i>Resource constraints</i>		
Pinned or copied string or array	12	No leak or double-free
Monitor	1	No leak
Global or weak global reference	247	No leak or dangling reference
Local reference	284	No overflow or dangling reference

Table 2. Classification and number of JNI constraints.

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

For each of the three classes, Figures 6, 7, and 8 summarize the state-machine representations, the entities to which they apply, and the errors they find. The remainder of this section describes in detail each constraint, its state machine representation, and how it detects errors.

5.1 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. Figures 6 shows the state machines for these three types of JVM constraints.

JNIEnv* state constraint. 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. When the program creates a native thread, Jinn learns about the JNIEnv* pointer from the JVM, and retrieves the thread ID from the operating system. It enters both into the state machine encoding, which is a map from thread ID to JNIEnv* pointer. Later, when a native thread calls any of the 229 JNI functions, Jinn looks up the expected JNIEnv* from the state machine encoding and compares it to the actual parameter of the call, reporting an error if the pointers differ.

Exception state 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

JNIEnv* state

Observed entity: A thread.

Error(s) discovered: JNIEnv* mismatch.

State machine encoding: Map from thread IDs to their expected JNIEnv* pointers.

State machine diagram: Trivial, omitted for brevity.

State transition	Language transition	Triggering functions
JNI call	Call:C→Java	Any JNI function e.g., CallVoidMethod

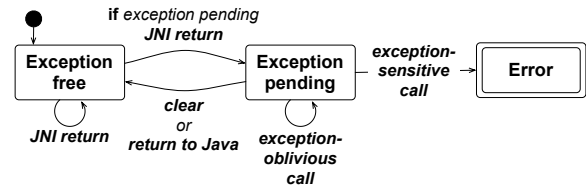
Exception state

Observed entity: A thread.

Error(s) discovered: Unhandled Java exception.

State machine encoding: Internal JVM structures.

State machine diagram:



State transition	Language transition	Triggering functions
JNI return	Return:Java→C	Any JNI function e.g., CallVoidMethod
Clear or return to Java	Return:Java→C Return:C→Java	ExceptionClear Return from any native method
Exception-oblivious call	Call:C→Java	Small set of clean-up functions e.g., ReleaseStringChars
Exception-sensitive call	Call:C→Java	All other JNI functions e.g., GetStringChars

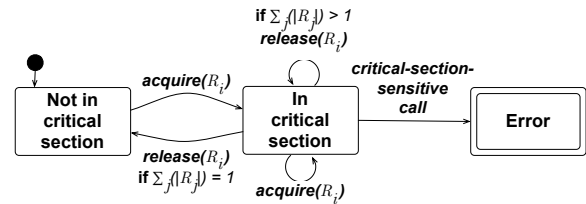
Critical-section state

Observed entity: A thread.

Error(s) discovered: Critical section violation.

State machine encoding: Map from a critical resource R_i to the number of times a given thread has acquired that resource.

State machine diagram:



State transition	Language transition	Triggering Functions
Acquire	Return:Java→C	GetStringCritical or GetPrimitiveArrayCritical
Release	Return:Java→C	ReleaseStringCritical or ReleasePrimitiveArrayCritical
Critical-section sensitive call	Call:C→Java	All other JNI functions e.g., CallVoidMethod

Figure 6. State machines for JVM state constraints.

Fixed typing

Observed entity: A reference parameter.

Error(s) discovered: Type mismatch between actual and formal parameter to JNI function.

State machine encoding: Map from entity IDs to their signatures.

State machine diagram: Trivial, omitted for brevity.

State transition	Language transition	Triggering functions
JNI call	Call:C→Java	JNI function defining a parameter with a fixed type, e.g., clazz parameter to CallStaticVoidMethod

Access control

Observed entity: A field ID.

Error(s) discovered: Assignment to final field.

State machine encoding: Map from field IDs to their modifiers.

State machine diagram: Trivial, omitted for brevity.

State transition	Language transition	Triggering functions
JNI call	Call:C→Java	Set<Type>Field or SetStatic<Type>Field

Entity-specific typing

Observed entity: A pair of ID parameters.

Error(s) discovered: Type mismatch for Java field assignment or between actual and formal of a Java method.

State machine encoding: Map from entity IDs to their signatures.

State machine diagram: Trivial, omitted for brevity.

State transition	Language transition	Triggering functions
JNI call	Call: C→Java	JNI function defining parameters with interrelated types, e.g., clazz and method in CallStaticVoidMethod

Nullness

Observed entity: A reference parameter.

Error(s) discovered: Unexpected null value passed to JNI function.

State machine encoding: None.

State machine diagram: Trivial, omitted for brevity.

State transition	Language transition	Triggering functions
JNI call	Call:C→Java	JNI function defining a parameter that must not be null, e.g., method parameter to CallStaticVoidMethod

Figure 7. State machines for type constraints.

results from the semantic mismatch in how C and Java handle exceptions. Any JNI call may lead to Java code that throws an exception, which causes a transition to the “exception pending” state when the JNI call returns. The JVM internally records this state transition for each Java thread, so Jinn does not need to interpose on JNI returns to track exception states, and can instead simply rely on the JVM-internal data structure for its state machine encoding. If the program returns from a JNI call and an exception is pending, the program must consume or propagate the exception. To do so, the programmer may first select from one of 20 *exception-oblivious* JNI functions that query the exception state and release JVM resources, before calling JNI’s `ExceptionClear` function. If the programmer calls any of the remaining *exception-sensitive* JNI functions while an exception is pending, Jinn intercedes and wraps the pending exception in an error report to the user.

Critical-section state constraints. JNI defines the phrase “JNI critical section” to describe a piece of C code that has direct access to a Java string or array, during which the JVM may take drastic measures such as disabling the garbage collector. A critical section starts with `GetStringCritical` or `GetPrimitiveArrayCritical`, and ends with the matching `ReleaseStringCritical` or `ReleasePrimitiveArrayCritical`. C code should hold these resources only for a short time. To prevent deadlock, C code must not interact with the JVM other than to acquire or release critical resources. In other words, during a critical section, C code must only call one of the four functions that get/release arrays/strings. We call these four functions critical-section *insensitive*, and all the remaining JNI functions critical-section *sensitive*. Jinn encodes the state machines by keeping, for each thread, a tally of the number of times that thread has acquired a specific critical resource. Jinn instruments the four “get” and “release” calls to manage these counts. Each acquisition of a resource R_i must be matched by a corresponding release. When the list of critical resources for a thread toggles between empty and non-empty, the critical-section state machine transitions correspondingly. Jinn interposes on all the 225 critical-section sensitive

functions to verify that the thread currently maintains no critical resources and that releases are well-matched.

Critical sections are tricky because they prohibit calls to most JNI functions, including those that Jinn uses for its own error checking. For example, Jinn does not check whether the argument to `ReleaseStringCritical` is in fact a Java string, since that would require calling `IsAssignableFrom` from within a critical region. At the same time, C code cannot exercise much JNI functionality while in a critical section and can legally call only four functions—to acquire more critical sections and to release them again.

5.2 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`. Consider 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 its safety, but if the program is unsafe at runtime, then the JVM throws an exception. In JNI, this code is expressed as follows.

```
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);
```

Since the C code expresses Java type information in strings, standard static type checking cannot resolve the types and even sophis-

ticated interprocedural analysis cannot always resolve them [10, 28]. Consequently, the C compiler does not statically enforce typing constraints on the “Collections” and “sort” names or the `ls` and `cmp` parameters. Furthermore, and unlike Java reflection, JNI does not even dynamically enforce typing constraints on the `clazz` and `method` descriptors.

In contrast, Jinn does enforce these and other JNI type constraints dynamically. Figure 7 summarizes the four constraints on types and the remainder of the section discusses them in detail. The figure omits the trivial state machines for brevity.

Fixed typing constraints. Type constraints require the runtime type of actuals to conform to the formals. For many JNI functions, the parameter type is, in fact, *fixed* by the function itself. For example, in `CallStaticVoidMethod(env, clazz, method, ls, cmp)`, the `clazz` actual must always conform to type `java.lang.Class`. We extracted this and comparable constraints by scanning the JNI header file for C parameters (e.g., `jstring`) with well-defined corresponding Java types (e.g., `java.lang.String`). We extracted additional fixed typing constraints from the informal JNI explanation in [19]. For example, `FromReflectedMethod` has a `jobject` parameter, whose expected type is either `java.lang.reflect.Method` or `java.lang.reflect.Constructor`. Overall, Jinn interposes on 151 JNI functions to verify 157 fixed typing constraints. For each check, Jinn obtains the class of the actual using `GetObjectType` and then checks compatibility with the expected type through `lsAssignableFrom`.

Entity-specific typing constraints. A plethora of JNI functions call Java methods or access Java fields. JNI references Java methods and fields via *entity 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 constrains the other parameters. In particular, the `clazz` must declare the method, and `ls` and `cmp` must conform to the formal parameters of the method. Jinn records method and field signatures upon return from JNI functions that produce method and field IDs. The entity ID constrains the types of method parameters or field values, and the receiver class (for static entities) or object (for instance entities), for each of 131 JNI functions that access a Java entity. When a program calls one of these functions that take an entity ID, Jinn interposes on the call to verify that the function conforms to the entity’s typing constraints.

Access control constraints. 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. The JNI specification is vague on legal accesses with respect to their visibility and final constraints. After some investigation, we found that in practice, 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, which may suppress Java access control when `setAccessible(true)` was successful. Honoring final is common sense. Despite the fact that reflection may mutate final fields, mutating them interferes with JIT optimizations, concurrency, and complicates the Java memory model. As with entity-specific typing, Jinn keeps track of field IDs, as well as which fields are final. Jinn raises an error if native code calls any of the 18 JNI functions that might assign to a final field.

Nullness constraints. Some JNI function parameters must not be null. For example, in `CallStaticVoidMethod(env, clazz, method, ls, cmp)`, the parameters `env`, `clazz`, and `method` must not be null. At the same time, some JNI functions do accept null parameters, for example, 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. We uncovered 416 non-null constraints among the 210 JNI functions that define parameters. Jinn reports to the user when the program violates any of these constraints.

5.3 Resource Constraints

A JNI resource is a piece of Java-related data that C code can acquire or release through JNI calls. For example, C code can acquire a Java string or array. Depending on the JVM implementation, the JVM either pins the string or array to prevent the garbage collector from moving it, or copies the array, and then passes C code a pointer to the contents. Other JNI resources include various kinds of opaque references to Java objects, which C code can pass to JNI functions, and which give C code some control over Java memory management. Finally, JNI can acquire or release Java monitors, which are a mutual-exclusion primitive for multi-threaded code.

APIs with manual or semi-automatic memory management suffer from well-known problems. (1) Section 3.2 illustrated one such problem: a use after a release corrupts JVM state through a dangling reference. There are three other common resource errors. (2) An acquire at insufficient capacity causes an overflow. (3) A missing release at the end of reference lifetime causes a leak. (4) A second release is a double-free. The Jinn analysis depends on the resource (e.g., array, string reference, object). In a few cases, Jinn cannot detect certain error conditions, because they are under-specified or hidden in C code. For instance, Jinn currently cannot detect when C code uses an invalid C pointer without calling a JNI function. In a few cases, Jinn need not check resource-related errors, since the JVM or other Jinn state machines already trap them. For example, when the JVM throws an `OutOfMemoryError` exception, Jinn already checks for correct exception handling.

While the state machines and error cases for all kinds of JNI resources are similar, they differ in the details due to the above reasons. Figures 8 shows these four different resource cases separately, and we now discuss each in more detail.

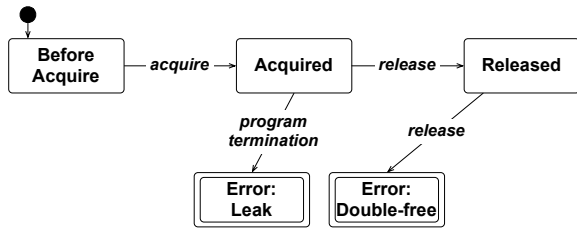
Pinned or copied string or array constraints. C code can temporarily obtain direct access to the contents of a Java string or array. JVMs may pin or copy the object to facilitate garbage collection. To make sure the JVM unpins the object or frees the copy, the C code must properly pair acquire/release calls. Jinn reports a leak for any resource that has not been released at program termination. Jinn reports a double-free for a resource it has already evicted from its state machine representation due to an earlier free. Jinn does not check for dangling references, because their uses happen in C code. Jinn does not check for overflow (i.e., an out-of-memory condition) in this state machine, because its exception checking subsumes this check.

Monitor constraints. A monitor is a Java mutual exclusion primitive. Jinn need not check overflow or double-free for monitors, since the JVM already throws exceptions. Jinn cannot check dangling monitors, since that requires divining when the programmer intended to release it. Jinn does report if a monitor is not released at program termination, which indicates a risk of deadlock.

Global reference or weak global reference constraints. A global or weak global reference is an opaque pointer from C to a Java object that is valid across JNI calls and threads. These references are explicitly managed, because the garbage collector needs to update them when moving objects and also treat global (but not weak) references as root. Jinn reports a leak for any unreleased global or weak global reference at program termination. Jinn reports a dangling reference error if the program uses a reference after a free. Double-free is a special case of the dangling reference error and overflow is a special case of Jinn’s exception state constraints.

Pinned or copied string or array

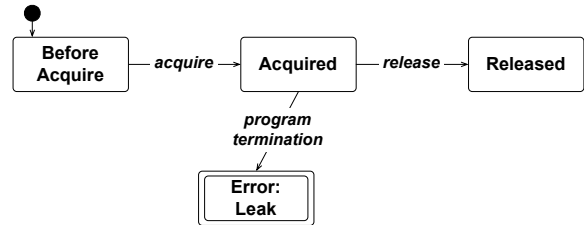
Observed entity: A Java string or array that is pinned or copied.
Error(s) discovered: Leak and double-free.
State machine encoding: A list of acquired JVM resources.
State machine diagram:



State transition	Language transition	Triggering functions
Acquire	Return:Java→C	Get<Type>ArrayElements and similar getter functions
Release	Return:Java→C	Release<Type>ArrayElements and similar release functions
Program termination		JVMTI callback

Monitor

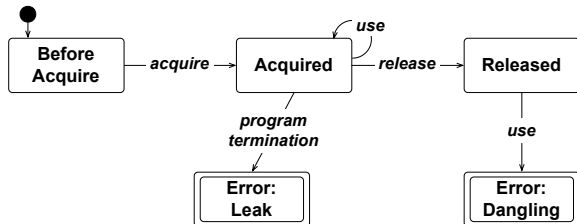
Observed entity: A monitor.
Error(s) discovered: Leak.
State machine encoding: A set of monitors currently held by JNI and, for each monitor, the current entry count.
State machine diagram:



State transition	Language transition	Triggering functions
Acquire	Call:C→Java	MonitorEnter
Release	Call:C→Java	MonitorExit
Program termination		JVMTI callback

Global reference or weak global reference

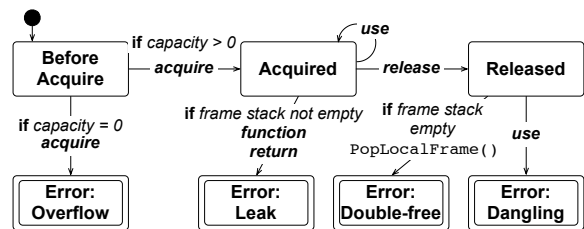
Observed entity: A global or weak global JNI reference
Error(s) discovered: Leak and dangling reference.
State machine encoding: A list of acquired global references.
State machine diagram:



State transition	Language transition	Triggering functions
Acquire	Return:Java→C	NewGlobalRef and NewWeakGlobalRef
Release	Return:Java→C	DeleteGlobalRef and DeleteWeakGlobalRef
Use	Call:C→Java	JNI function taking reference e.g., CallVoidMethod
	Return:C→Java	Native method returning reference, e.g., Class.getClassContext
Program termination		JVMTI callback

Local reference

Observed entity: A local JNI reference
Error(s) discovered: Overflow, leak, dangling, and double-free.
State machine encoding: For each thread, a stack of frames. Each frame has a capacity and a list of local references.
State machine diagram:



State transition	Language transition	Triggering functions
Acquire	Call:Java→C	Native method taking reference
	Return:Java→C	JNI function taking reference e.g., GetObjectField
Release	Return:Java→C	DeleteLocalRef or PopLocalFrame
	Return:C→Java	Return from any native method
Use	Call:C→Java	JNI function taking reference e.g., CallVoidMethod
	Return:C→Java	Native method returning reference, e.g., Class.getClassContext

Figure 8. State machines for resource constraints.

Local reference constraints. 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 reference into its state machine encoding upon acquire

and removes it upon release. Jinn performs bookkeeping to support overflow checks, since 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 four error cases as follows. (1) Jinn detects *overflow* if the current local frame exceeds 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) Jinn detects a *double-free* when `DeleteLocalRef` is called twice for the same reference, or if nothing is left to pop on a call to `PopLocalFrame`.

6. 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.

6.1 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 9.10 on the Linux 2.6.31-19 kernel.

JNI programs. We used several JNI programs: microbenchmarks, SPECjvm98 [23], DaCapo [6], Subversion 39004 (2009-08-31), Java-gnome-4.0.10, and Eclipse 3.4. The microbenchmarks are a collection of 16 small JNI programs, which are designed to trigger one each of the error states in the eleven state machines shown in Figures 6, 7, and 8. The microbenchmarks also cover all pitfalls in Table 1 with exception of Pitfall 8, which cannot be detected at the language boundary. SPECjvm98 and DaCapo are written in Java, but exercise native code in the system library. Subversion, Java-gnome, 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 30-100 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.

6.2 Performance

This section evaluates the performance of Jinn. Table 3 shows the results. Jinn adds instructions to every language transition between the JVM and native libraries, interposing and checking transitions. The second column counts the total number of transitions between Java and C in the system libraries using HotSpot. The third column shows the execution times of runtime checking for HotSpot. Execution times are normalized to production runs of HotSpot without any dynamic checking. The fourth column reports Jinn’s framework overhead due to interposition on language transitions. The fifth column reports the total time, which includes state machine encoding, transitions, and error checking. On average, Jinn has a modest 14% execution time overhead and most of the overhead (all but 4%) comes from runtime interposition, rather than executing the analysis code.

6.3 Coverage of Jinn and JVM Runtime Checking

This section shows that Jinn covers qualitatively and quantitatively more JNI bugs than the state-of-art dynamic checking in production JVMs.

Quality. We run the 16 microbenchmarks with HotSpot, J9, and Jinn. Figure 9 compares their error messages on the representative *ExceptionState* microbenchmark, which violates the exception

Benchmark	Language transition counts	Normalized execution times		
		Runtime checking	Interposing	Jinn Checking
antlr	441,789	1.04	0.98	1.05
bloat	839,930	1.02	1.19	1.20
chart	1,006,933	1.02	1.08	1.12
eclipse	8,456,840	1.01	1.17	1.20
fop	1,976,384	1.07	1.14	1.37
hsqldb	206,829	0.88	1.04	1.05
kython	56,318,101	1.03	1.10	1.16
luindex	1,339,059	1.03	1.08	1.13
lusearch	4,080,540	1.04	1.09	1.21
pmd	967,430	1.04	1.10	1.13
xalan	1,114,000	1.01	1.17	1.19
compress	14,878	0.98	1.09	1.08
jess	153,118	0.99	1.22	1.17
raytrace	29,977	1.04	1.16	1.14
db	133,112	0.99	1.01	1.02
javac	258,553	1.06	1.16	1.14
mpegaudio	46,208	1.00	1.01	1.04
mtrt	32,231	1.01	1.11	1.14
jack	1,332,678	1.04	1.10	1.21
GeoMean		1.01	1.10	1.14

Table 3. Jinn performance on SPECjvm and DaCapo with HotSpot.

state constraints of Section 5.1. The C code in the benchmark ignores an exception raised by 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 both illegal JNI calls, their calling contexts, and the source location of the original Java exception. In addition to precise reports, Jinn’s error reporting integrates with debuggers. Java debuggers like `jdb` and Eclipse JDT can catch the custom exception, and programmers can then inspect the Java state to find the failure’s cause. Even better, the Blink Java/C debugger [17] can present the entire program state, including the full calling context consisting of both Java and C frames.

Quantity. The behavior of the production runs without dynamic checkers ranges from ignoring the bug to simply crashing to raising a null pointer exception—none of which is correct. The dynamic checkers built into the HotSpot and J9 JVMs also behave inconsistently in more than half of our microbenchmarks (9 of 16). Jinn is the only dynamic bug-finder that consistently detects and reports the JNI bugs in our 16 microbenchmarks by throwing an exception. Quantitative coverage of Jinn, HotSpot, and J9 is 100%, 56%, and 50%, respectively, with exceptions, warnings (print to console and keep running), and errors (print to console and terminate) counting as valid bug reports. Jinn’s 100% coverage on our own, specifically designed test suite is hardly surprising and does not imply that Jinn catches all JNI bugs. But the low JVM coverage demonstrates that error checking in previous practice was at best incomplete. Furthermore, JNI constraint violations are common and well-documented [10, 11, 16, 17, 18, 26, 27], underlining the need for better constraint enforcement.

6.4 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

```

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

```

(a) HotSpot

```

JVMJNCK028E JNI error in GetMethodID: This function
cannot be called when an exception is pending
JVMJNCK077E Error detected in ExceptionState.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 ExceptionState.call(Native Method)
at ExceptionState.main(ExceptionState.java:5)
Caused by: jinn.JNIAssertionFailure:
An exception is pending in GetMethodID.
... 3 more
Caused by: java.lang.RuntimeException:
checked by native code
at ExceptionState.foo(ExceptionState.java:9)
... 2 more

```

(c) Jinn

Figure 9. Representative JVM and Jinn error messages using a microbenchmark that violates the *exception state* constraint.

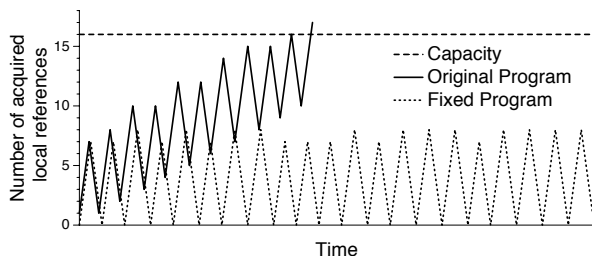


Figure 10. Time-series of acquired local references with leak and its fix in the fourth execution of a Java native method: `Java_org_tigris_subversion_javahl_SVNClient_info2`.

these open-source programs are in wide industrial and academic use with a long revision history. These case studies show that Jinn finds errors in widely-used programs.

6.4.1 Subversion

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

Overflow of local references. Jinn found that Subversion allocated more than 16 local references in two call sites to JNI functions: line 99 in `Outputer.cpp` and line 144 in `InfoCallback.cpp`. Figure 10 compares the time-series of acquired local references for the original and the fixed program. The original program overflows the pool of 16 local references without requesting more capacity—as

detected by Jinn when acquiring yet another local reference. One reported 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 passes the regression test even under Jinn, since the number of active local references never exceeds 8. This overflow did not crash HotSpot and J9, but represents a time bomb. A highly optimized JVM may crash if it assumes that JNI code is well-behaved and eliminates bound checking of the bump pointer for local references.

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 crash production JVMs. To understand this better, 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, rendering the fact that the actual is a dangling reference irrelevant. If other JVMs are implemented similarly, this bug will remain hidden. But the code again 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 internally represent strings in UTF8 format as proposed by Zilles [30] and then share them directly with JNI.

6.4.2 Java-gnome

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

Nullness. Jinn reports a bug identified in the Blink debugger paper [17]. Note, however, that Blink requires running the Java program in a full-fledged debugger, while Jinn is a light-weight dynamic checker.

Use of dangling local reference. Jinn reports and diagnoses bug 576111 for the Java-gnome project, which violates a constraint on semi-automatic resources. Jinn reports that Line 348 of `binding_java_signal.c` violates a local reference constraint.

```

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

```

The `bjc->receiver` is a dead local reference. A Java-gnome developer confirmed the diagnosis. This bug did not crash HotSpot and J9, but, as noted before, bugs that are only benign due to implemen-

tation characteristics of a specific JVM vendor are time bombs and should be fixed.

6.4.3 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 callback control and a Java inner class. Because the production JVM may not use the object value, this bug has survived multiple revisions.

6.5 Limitations

While Jinn consistently detects more JNI bugs than previous work, it currently falls short on three categories of bugs: (1) constraint checking requires currently forbidden JNI functionality, (2) “correctness” is not black-and-white, and (3) checks at the language boundary are insufficient. The primary example for Category 1, JNI functionality being forbidden, are the JNI critical sections discussed in Section 5.1, which prevent Jinn from calling `IsAssignableFrom` to enforce type constraints. An example for Category 2, correctness gray-zone, is whether or not Jinn should report C code accessing private fields of Java objects. The JNI specification is silent about this, and accessing private Java fields from C is an entrenched practice. Another gray-zone example is whether or not a piece of code must be guarded by a monitor. After all, one person’s race condition is another person’s optimization. An example for Category 3, checks at the language boundary being insufficient, is Pitfall 8 from Table 1. JNI does not terminate Unicode strings with NULL characters when returning them to C. Since C code subsequently manipulates the string data directly without calls through JNI functions, correctness checking at the language boundary as done by Jinn is insufficient. It instead requires checking C memory safety as performed, for instance, by SafeJNI [25]. Another example for Category 3 is the currently omitted “dangling” state of the state machine for a pinned or copied string or array (see Figure 8). As before, since C code manipulates the string or array directly, Jinn’s language boundary checks are insufficient.

7. Generalization

This section demonstrates that our technique generalizes to other languages by applying it to Python/C 2.6 [1]. We first discuss the similarities and differences between JNI and Python/C. We then present a synthesized dynamic checker for Python/C’s manual memory management. We leave to future work the full specification of Python/C FFI constraints and the complete implementation of a dynamic analysis for these constraints.

7.1 Python/C Constraint Classification

Like JNI, the Python/C specification describes numerous rules that constrain how programmers can combine Python and C. These constraints fall into the same classes from Section 5: (1) interpreter state constraints, (2) type constraints, and (3) resource constraints.

State constraints. Python/C constrains the behavior of exceptions and threads. Python/C’s exception constraints mirror those of JNI: C code should immediately handle the exception or propagate it back to Python. While not explicitly stated in the manual, these constraints also imply that native code should not invoke

```
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("s", "sssss"),
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 11. Python/C dangling reference error. The borrowed reference `first` becomes a dangling reference when `pythons` dies.

other Python/C functions while an exception is pending. For thread constraints, Python/C differs slightly from JNI 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.

Python/C permits C code to release and re-acquire the GIL around blocking I/O operations. It also 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. As a result, Python/C requires bookkeeping for the GIL similar to that for JNI critical sections discussed in Section 5.1.

Type constraints. Because Python is dynamically typed, types in Python/C are less constrained than in JNI. The Python interpreter performs dynamic type checking for many operations on built-in types. However, sometimes the interpreter forgoes these type checks—as well as some null checks—for performance reasons. Consequently, if a program passes a mistyped value to a Python/C call, the program may crash or exhibit undefined behavior. A dynamic analysis based on the type constraints of Section 5.2 would enable reliable detection of these errors, at the cost of reintroducing dynamic checking for some Python/C functions.

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

7.2 Synthesizing Dynamic Checkers

To ensure that FFI programs correctly use co-owned and borrowed references, we implemented a use-after-release checker for Python/C’s reference counting memory management.

Example memory management error. Figure 11 contains an example Python/C function that mismanages its references. The refer-

ence first in Line 6 is borrowed from the reference pythons. When Line 8 decrements the reference count for pythons, the reference dies. The Python/C 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 Python’s semantics are undefined for such a case. In practice, Figure 11’s behavior depends on whether the interpreter reuses the memory for first between the implicit release in Line 8 and the explicit use in Line 10.

Synthesizer and generated checker. Our synthesizer takes a specification file that lists which functions return new or borrowed references. The generated checker detects memory management errors by tracking co-owned references and their borrowers. For example, the checker determines that pythons is a co-owned reference and that first borrows from pythons. When a co-owner relinquishes a reference by decrementing its count, all its borrowed references become invalid. If the program uses an invalid borrowed reference, as Figure 11 does on Line 10, then the checker signals an error.

Interposing on language transitions. Integrating the dynamic analysis with Python/C is more challenging than for JNI. Python lacks an interface comparable to the JVM tools interface, thus requiring that the dynamic analysis be statically linked with the interpreter. Furthermore, Python/C bakes in some of the Python interpreter’s implementation details, which makes the API less portable than JNI and complicates interposing on language transitions. In particular, (1) Python/C makes prevalent use of C macros, (2) the Python interpreter internally uses Python/C functions, and (3) some variadic functions lack non-variadic counterparts.

Python/C makes extensive use of C macros. Some macros directly modify interpreter state without executing a function call. Because Python/C does not execute a function, our dynamic analysis has nothing to interpose on and cannot track the behavior. We overcame this limitation by replacing the macros with equivalent functions. This change requires programmers to re-compile their native code extensions against our customized interpreter, but it does not require them to change their code.

Because the Python interpreter internally calls Python/C functions, the dynamic analysis cannot easily detect application-level language transitions. Even if it could detect such transitions, function interposition and transition detection would significantly slow down the interpreter. We overcame this limitation by creating an interpreter-only copy of every Python/C function. We then used automatic code-rewriting to make the interpreter call the unmodified copies. Our dynamic analysis interposes on the originals, which are used by native code extensions.

A *variadic* C function such as `printf` takes a variable number of arguments. Our synthesizer interposes on each variadic function by wrapping it with code that calls an equivalent, non-variadic version of the function such as `vprintf`. Python does not provide non-variadic equivalents for all its variadic functions; where necessary, we added the non-variadic equivalents.

Despite these implementation challenges, our Python/C dynamic analysis substantially follows from our JNI dynamic analysis, thus demonstrating the generality of our approach. Both FFIs have large numbers of constraints that fall into three classes. Both FFIs also support specifying the constraints as state machines, mapping state machine transitions to language transitions, and then applying the state machines to program entities.

8. Conclusion

This paper seeks to improve the correctness of multilingual programs using thorough FFI specification and dynamic analysis. We identify three classes of FFI constraints and shows how to encode them in about a dozen state machines. The three classes capture the

key semantic mismatches that multilingual interfaces must negotiate. The state machines, in turn, capture the complete constraints for correctly using such interfaces. We also show how to use synthesis for mapping the state machine specifications into context-sensitive dynamic bug checkers inserted a language transitions. Notably, we generate dynamic bug checkers for JNI and Python/C. We show that Jinn, the synthesized bug checker for JNI, uncovers previously unknown bugs in widely-used Java native libraries. Our approach to multilingual constraint representation, constraint generation, and FFI usage verification is the most concise, practical, and effective on to date.

Acknowledgments

We thank our shepherd Alex Aiken and the anonymous reviewers for their feedback on this paper. We thank Mike Bond for 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.

This work is supported by NSF CNS-0448349, NSF CNS-0615129, NSF SHF-0910818, NSF CSR-0917191, NSF CCF-0811524, NSF CNS-0719966, NSF CCF-0448128, NSF CCF-0724979, Samsung Foundation of Culture, and CISCO. Any opinions, findings, and conclusions expressed herein are the authors’ and do not necessarily reflect those of the sponsors.

References

- [1] Python/C API reference manual. Python Software Foundation, <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 *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 345–364, 2005.
- [3] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 143–162, 2008.
- [4] T. Ball and S. K. Rajamani. SLIC: a specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, Jan. 2002.
- [5] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *USENIX Tcl/Tk Workshop (TCLTK)*, pages 129–139, 1996.
- [6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dinkelage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.
- [7] F. Chen and G. Rosu. MOP: An efficient and generic runtime verification framework. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 569–588, 2007.
- [8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ACM International Conference on Software Engineering (ICSE)*, pages 411–420, 1999.
- [9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Oct. 2000.

- [10] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 62–72, 2005.
- [11] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *European Symposium on Programming (ESOP)*, pages 309–324, 2006.
- [12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, June 2005.
- [13] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 19–38, 2007.
- [14] A. Kaplan, J. Bubba, and J. C. Wileden. The Exu approach to safe, transparent and lightweight interoperability. In *IEEE International Computer Software and Applications Conference (COMPSAC)*, page 393, 2001.
- [15] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, Apr. 1988.
- [16] G. Kondoh and T. Onodera. Finding bugs in Java native interface programs. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 109–118, 2008.
- [17] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley. Debug all your code: Portable mixed-environment debugging. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 207–226, 2009.
- [18] S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *ACM Conference on Computer and Communications Security (CCS)*, pages 442–452, 2009.
- [19] S. Liang. *The Java Native Interface: Programmer’s Guide and Specification*. Addison-Wesley, 1999.
- [20] NaturalBridge. BulletTrain JNI Checking Examples. http://web.archive.org/web/*/http://www.naturalbridge.com/jnichecking.html, Jan. 2001.
- [21] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, 2002.
- [22] T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit. Automatic generation of library bindings using static analysis. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 352–362, 2009.
- [23] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.
- [24] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, Feb. 2000.
- [25] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe Java native interface. In *IEEE International Symposium on Secure Software Engineering (ISSSE)*, pages 97–106, 2006.
- [26] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In *Usenix Security Symposium (SS)*, pages 365–377, 2008.
- [27] G. Tan and G. Morrisett. ILEA: Inter-language analysis across Java and C. In *ACM Conference on Object-Oriented Programming Systems and Applications (OOPSLA)*, pages 39–56, 2007.
- [28] Z. Tatlock, C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Deep typechecking and refactoring. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 37–52, 2008.
- [29] The GNOME Project. GNOME bug tracking system. Bug 576111 was opened 2009-03-20. <http://bugzilla.gnome.org>.
- [30] C. Zilles. Accordion arrays: Selective compression of unicode arrays in Java. In *ACM International Symposium on Memory Management (ISMM)*, pages 55–66, 2007.