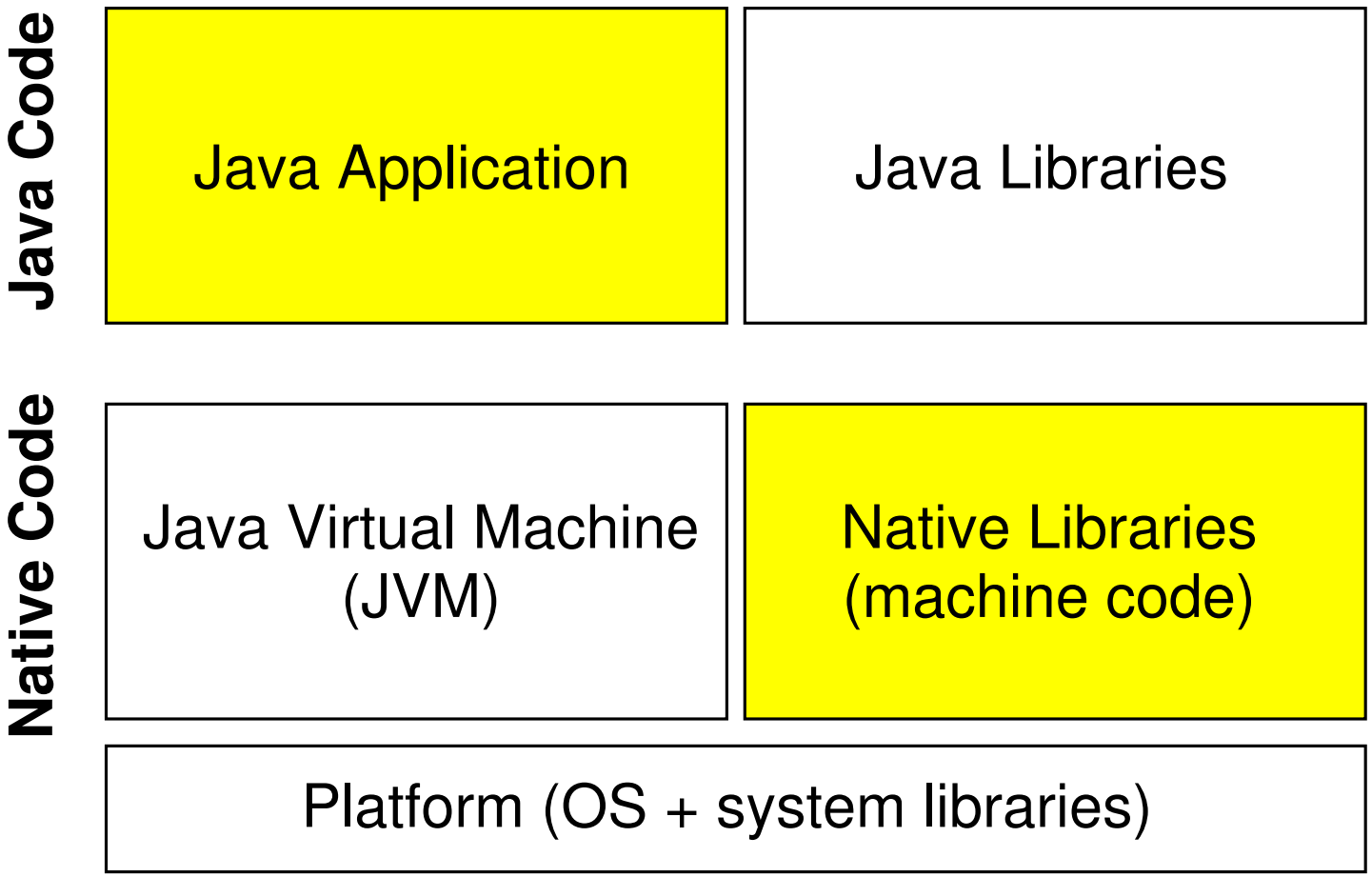
Extraction of Properties in C Implementations of Security APIs for Verification of Java Applications

Cyrille Artho, Yutaka Oiwa, Kuniyasu Suzaki
Research Center for Information Security (RCIS), AIST,
Tokyo, Japan

Masami Hagiya
University of Tokyo, Tokyo, Japan

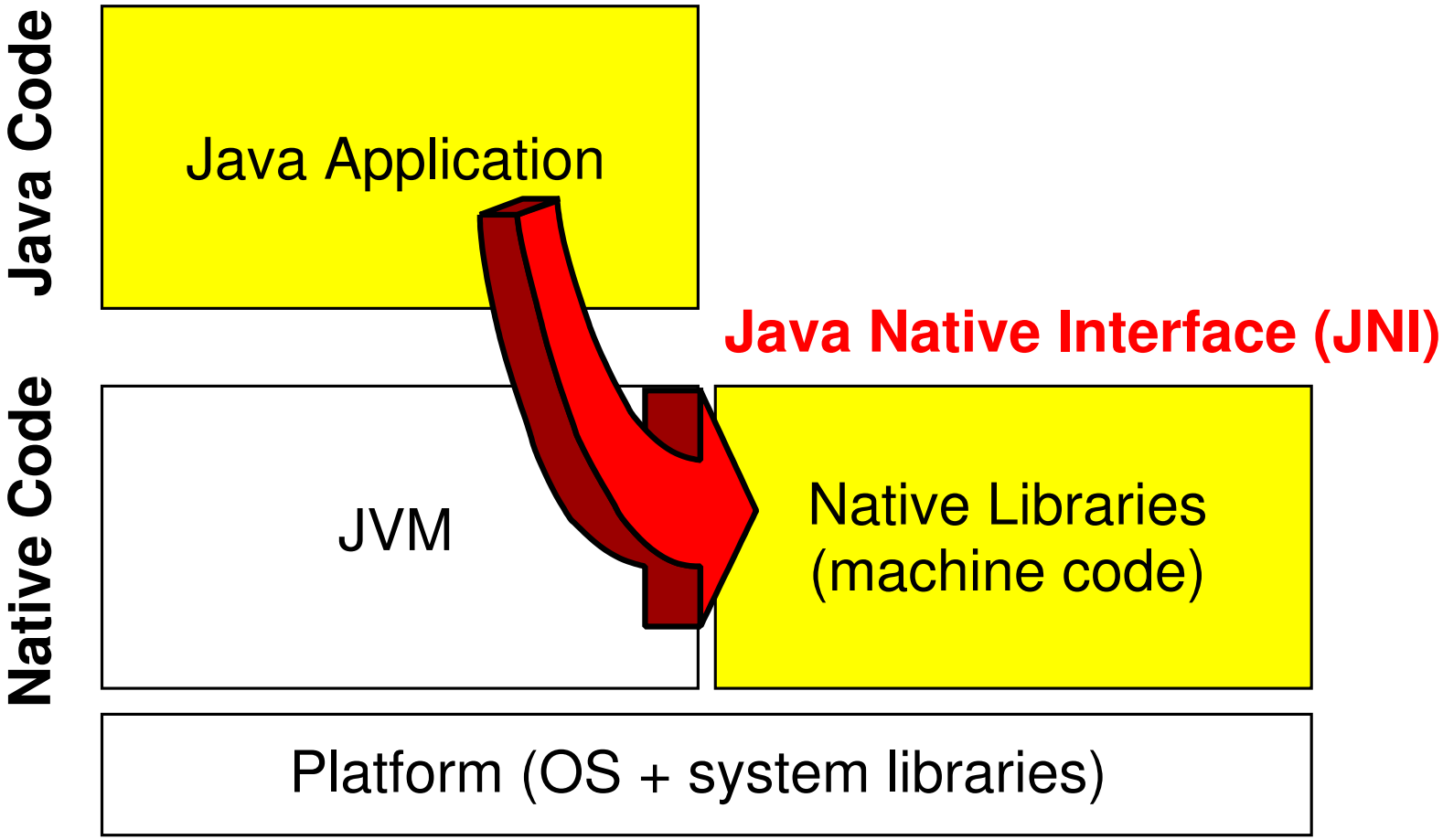
07/11/2009

Anatomy of a Java application



Java code can interact with native libraries via Java Native Interface (JNI).

Java Native Interface (JNI)



Effects of JNI calls difficult to analyze!

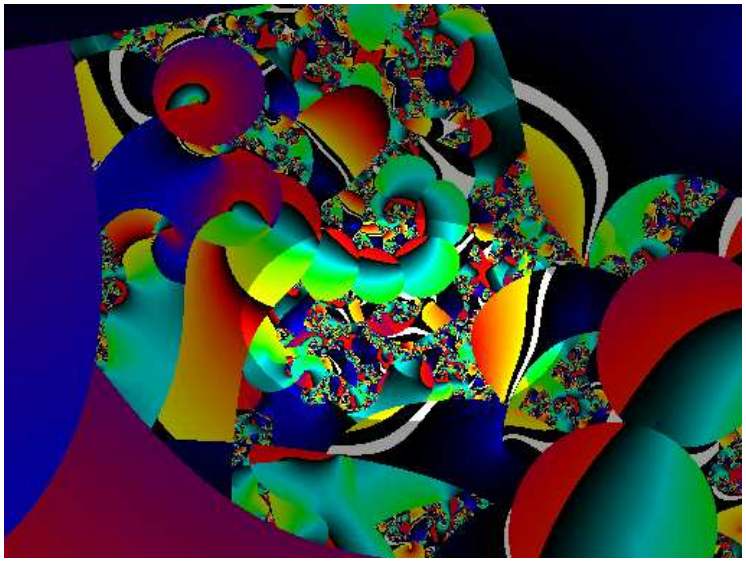
Overview

1. **Dynamic and static program analysis.**
2. Security properties in JNI code.
3. Bridging the gap between Java and native code.
4. Conclusion.

Dynamic and Static Analysis

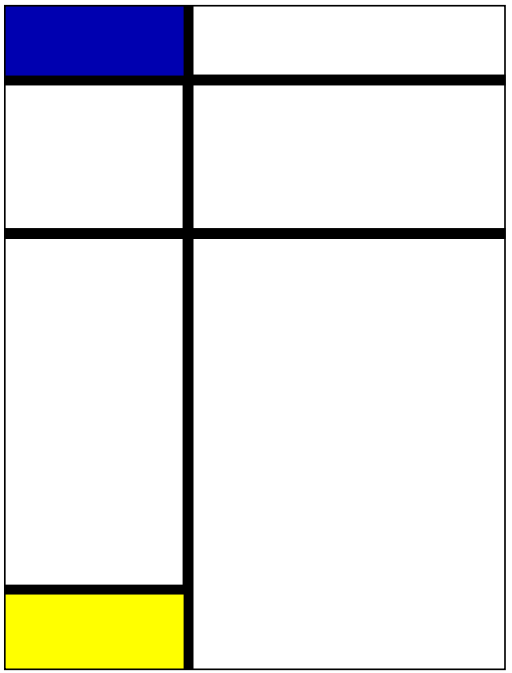
Dynamic Analysis

- “at run time”
- analyze real system



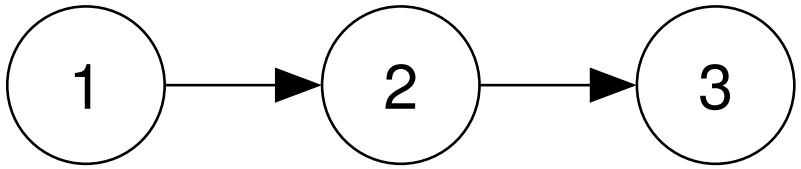
Static Analysis

- “at compile time’
- analyze simplified system (model)



Two divided worlds?

Run-time Verification

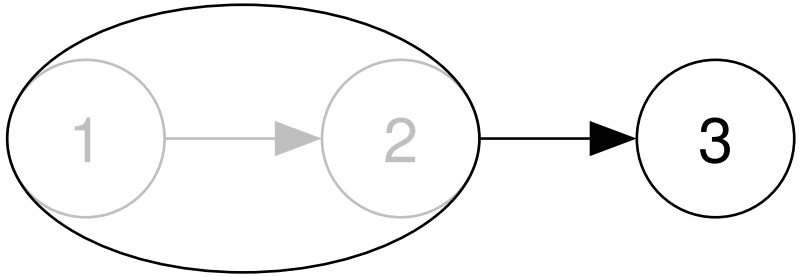


- Concrete values/states
- Full-sized system
- Testing never exhaustive
- Dependent on tests + schedule

May miss errors

Real example scenario

Static Analysis



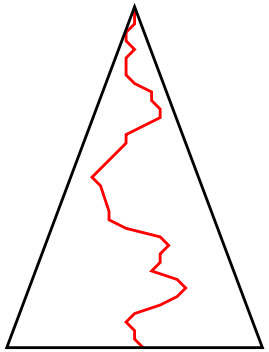
- Abstract values/states
- Smaller system
- Complete exploration possible
- Requires precise pointer analysis

Exhaustive search possible

False warnings

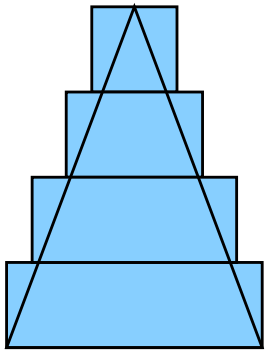
Strengths and weaknesses

Run-time Verification (RV)



- Single event trace
- Depends on schedule
- **Unsound**
- **Precise information**

Static Analysis

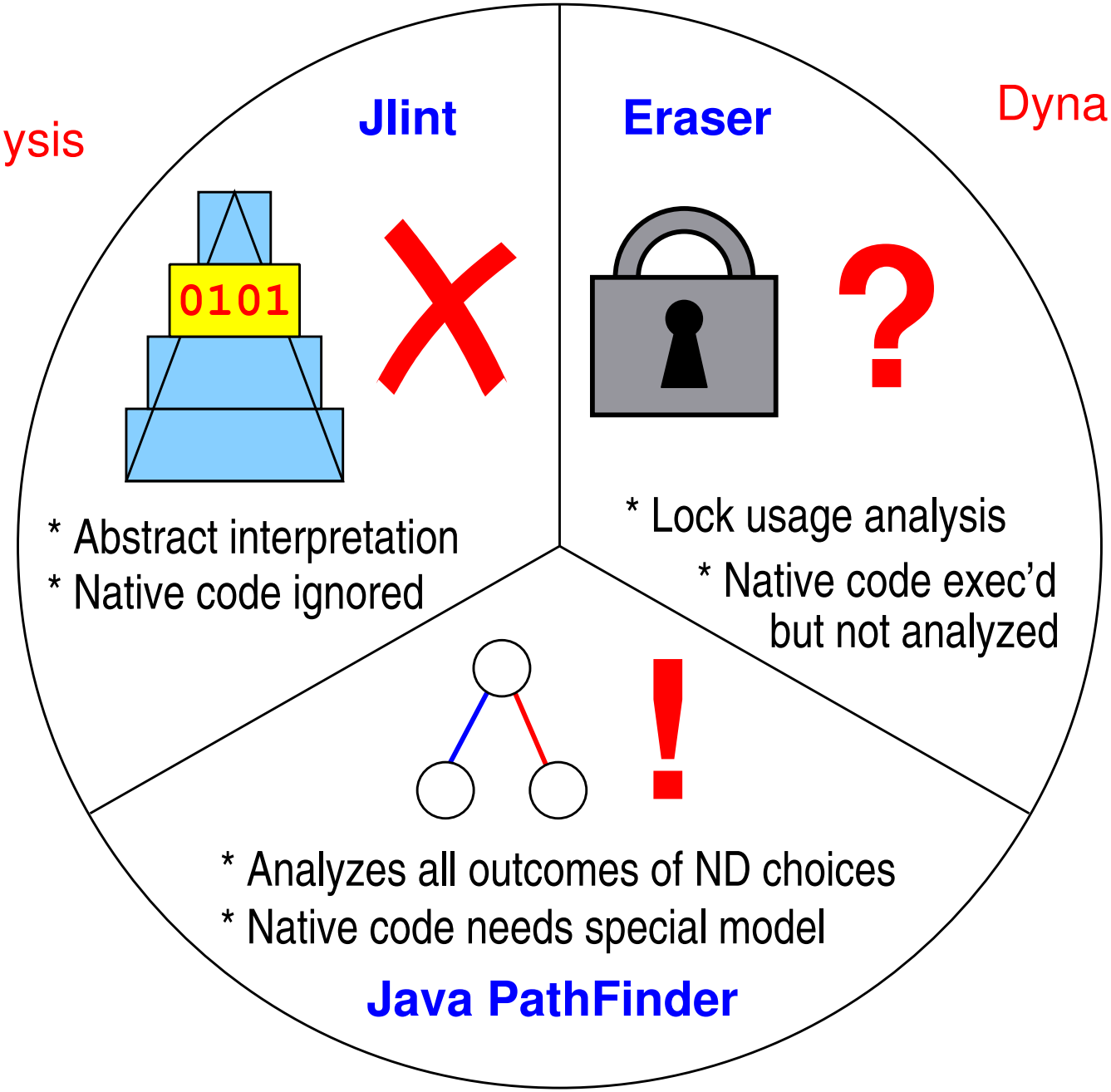


- Over-approximation
- Depends on abstraction
- **Sound**
- **Requires manual tuning**

Complementary technologies

- Tools typically support only one given programming language.

Verification platforms for Java and native code



Overview

1. Dynamic and static program analysis.
2. **Security properties in JNI code.**
3. Bridging the gap between Java and native code.
4. Conclusion.

Platform mismatch between application and library

Layer	Language	Description
Application	**Java**	**Target of this project**
Java library	**Java**	**High-level functions**
JNI layer	Java	passes library calls to low-level code
JNI impl.	C	C counterpart of JNI
Crypto library	**C**	**Low-level functions**
Device driver	C	(If present) interface to hardware

- Many crypto/security functions are implemented as a C library.
- Existing Java tools only analyze Java code (top three layers).
- How much information from lower layer(s) can we bring to the top?

Example

Java declaration:

```
public final static native void
TPM_NONCE_nonce_set(long jarg1, TPM_NONCE jarg1_, short[] jarg2);
```

C code:

```
SWIGEXPORT void JNICALL
Java_iaik_tc_tss_impl_jni_tsp_TspiWrapperJNI_TSS_1NONCE_1nonce_1set
(JNIEnv *jenv, jclass jcls, jlong jarg1, jobject jarg1_,
 jshortArray jarg2) {
    // other declarations omitted
    if (jarg2 && (*jenv)->GetArrayLength(jenv, jarg2) !=
        TPM_SHA1BASED_NONCE_LEN) {
        SWIG_JavaThrowException(jenv,
                                SWIG_JavaIndexOutOfBoundsException,
                                "incorrect array size");
        return;
    }
...
```

Goal of this project

Layer	Language	Description
Application	**Java**	**Target of this project**
Java library	**Java**	**High-level functions**
JNI layer	Java	passes library calls to low-level code
Crypto library	**C → Java**	**Low-level functions**

Convert native method to Java code:

```
public final static void
TPM_NONCE_nonce_set(long jarg1, TPM_NONCE jarg1_, short[] jarg2) {
    if ((jarg2 != null)
         && (jarg2.length != TPM_SHA1BASED_NONCE_LEN) {
        throw new IndexOutOfBoundsException("incorrect array size");
        return;
    }
}
```

Benefits

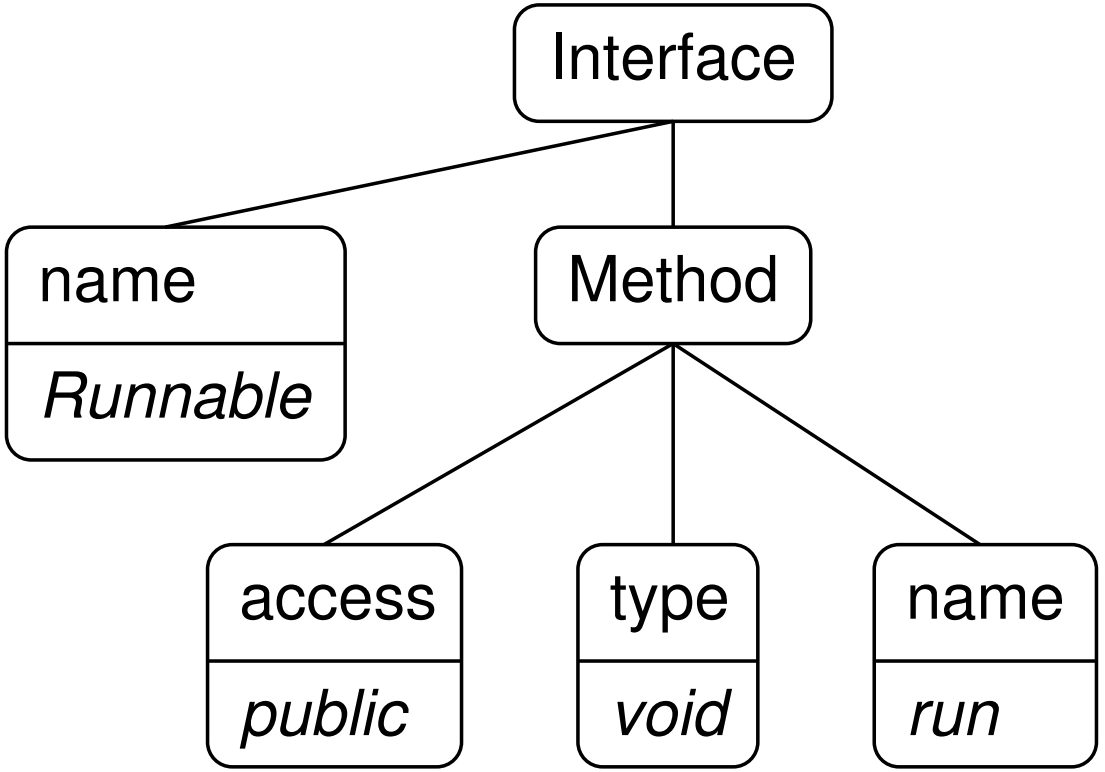
- Better integration into the analysis tool.
- Combining properties of multiple implementations.
- Usage of various analysis techniques:
  - static analysis
  - symbolic execution
  - model checking
  - fault injection
- Cross-platform tools do not exist (yet)!

How to bring C code into the Java world?

Overview

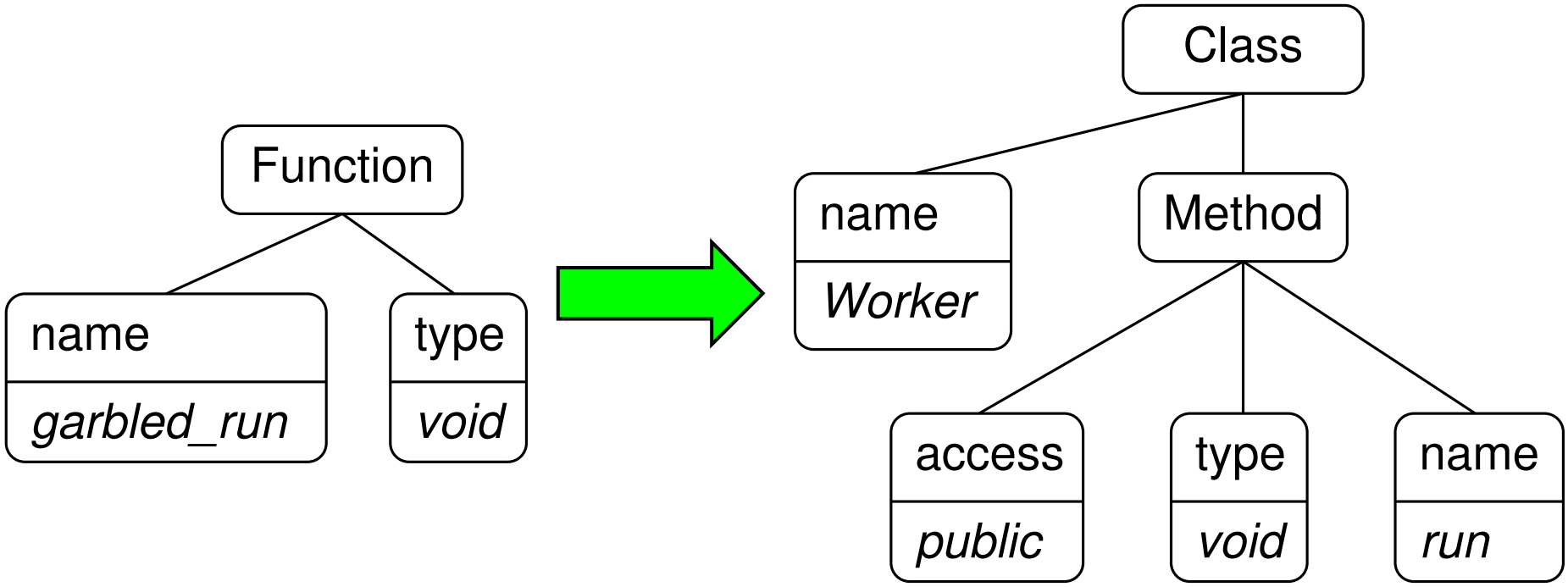
1. Dynamic and static program analysis.
2. Security properties in JNI code.
3. **Bridging the gap between Java and native code.**
4. Conclusion.

Model-driven architecture



- Domain-specific language models problem data.
- Domain = programming languages.
- Data = Abstract Syntax Tree.

Implementation architecture



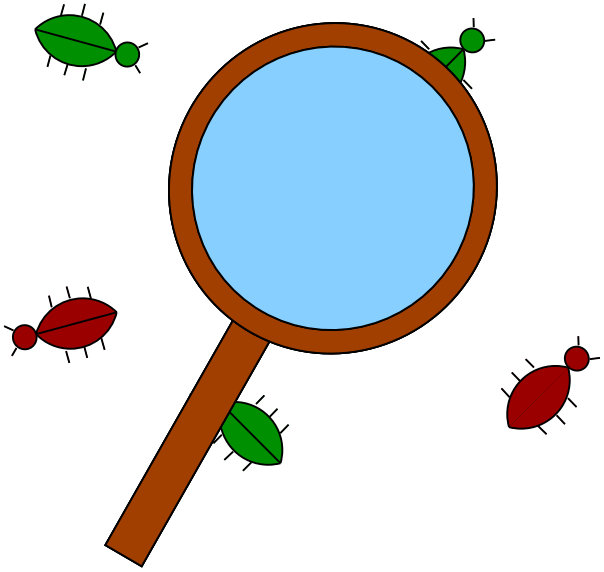
- Create same representation for C and Java code [Gondow, Maruyama].
- Map JNI „name mangling” to Java name.
- Map C code constructs as far as applicable.

Why is a partial code mapping useful?

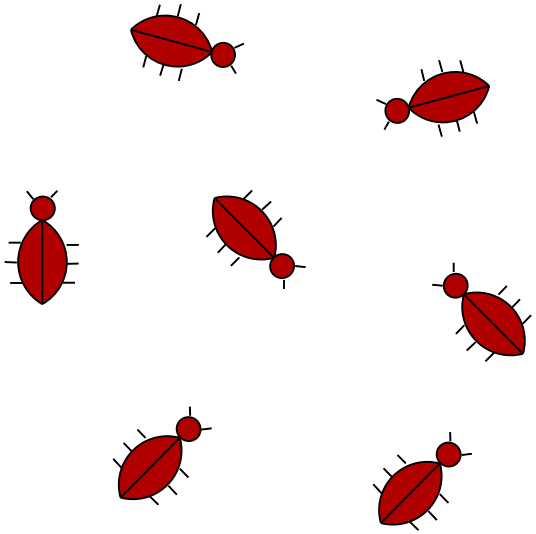
- Analyze Java **Application**.
- Are security APIs used correctly?
  - Under all circumstances: static analysis.
  - Create test cases: symbolic execution.
  - Analyze multi-threading: model checking.
  - Exception handling: fault injection.
- **Implementation** of APIs not relevant.
  - Design by contract: preconditions/postconditions important.
  - Implementation details can be verified by other tools (in C).

Conclusion

Java application



C library

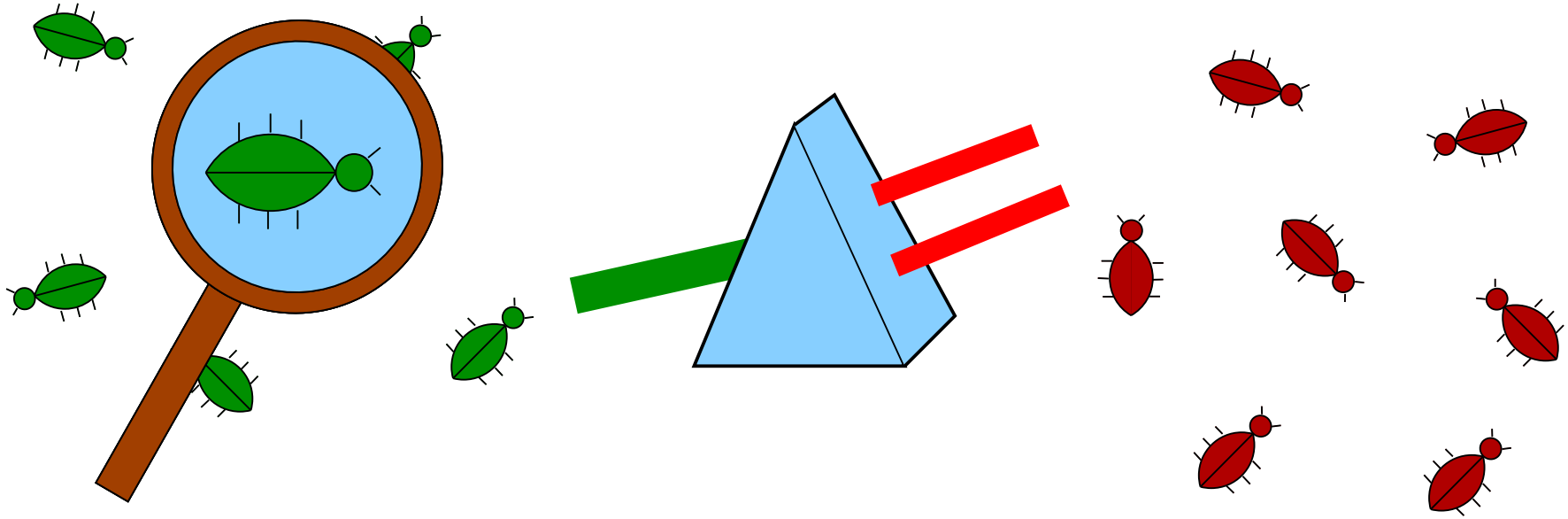


Java analysis tools miss C bugs!

Conclusion

Java application

C library



Make C library usage bugs visible!