Extraction of properties in C implementations of security APIs for the verification of Java applications

Cyrille Artho* Yutaka Oiwa* Kuniyasu Suzaki* Masami Hagiya†

Abstract

Java applications utilize various security APIs for cryptography and access control, such as available through packages java.security and javax.crypto. For performance reasons, these libraries internally use an implementation written in C, accessed through the Java Native Interface. Our goal is to extract properties in the source code of the C library, and translate these assertions back into the Java domain. This allows these properties to be used in verification of Java code, opening up various applications that are not possible when verifying binary code.

Keywords: Software model checking, software testing, Java, C, API, verification, security

1 Introduction

Java offers various security and cryptography facilities [11]. Security includes permission management (access control managed by security policies) and secure class loading. Cryptography includes access to cryptographic functions, such as hashing and encryption ciphers, and cryptographic protocols [16].

Java applications request security services from the Java platform. In order to accommodate for differences between different platforms, implementations are typically not provided by the standard library itself. Instead, they are accessed via *providers,* which encapsulate algorithm-specific implementations behind a standardized application programming interface (API). This allows an application to be independent of a provider. The same kind of function can be implemented by different providers, and exchanged if necessary [16].

For instance, certain access control features may only be available if the underlying operating system or file system supports corresponding mechanisms. Secure class loading may take advantage of specific hardware (trusted computing) [7, 21]. Cryptographic functions implemented in hardware or software may have to be replaced by different implementations if an existing implementation is too slow [1], or if an underlying cryptographic algorithm has been broken, i. e., demonstrated not to be cryptographically secure anymore [9].

Because the underlying functionality and API are very system-specific and low-level, the software library is usually implemented in C. Table 1 shows how in a Java application, access to C code is provided by the Java Native Interface (JNI) [11, 17]. This causes a problem when a Java application is analyzed by Java-specific tools, such as software model checkers [22]. An analysis tool may not be able to inspect or execute C code. In model checking, the inability to handle actions outside the given bytecode platform is a well-known problem when analyzing application that use library functionality such as network communication [5]. Because the effect of C code cannot be controlled by the Java platform, analysis tools may provide incorrect results. For model checking, a tool has to be able to restore the entire program state to a previous state. If C code is involved, side-effects of its execution may prove to be impossible to revert. This effectively puts programs using JNI calls outside the range of model checkers.

We plan to extract properties from low-level C code that are relevant for the correct behavior of Java applications. In this way, we can apply various inspection and analysis techniques to Java programs that are not possible otherwise.

Compared to similar work [19, 18], we plan to generate executable code instead of property annotations. We think that existing toolkits for code analysis that represent the abstract syntax tree as XML data may be the appropriate platform for a unified representation of the data [10, 14]. Mapping rules can then relate C code fragments to Java code.

*Research Center for Information Security (RCIS), AIST, Tokyo, Japan
†University of Tokyo, Tokyo, Japan

Table 1: Architecture of Java application using Java library backed by native code.

Layer	Language	Description
Application	Java	Written by developers, target of verification in this project
Java library	Java	High-level functions (e. g., security and cryptography)
JNI layer	Java	Java Native Interface: passes library calls to low-level code
JNI impl.	C	C counterpart of JNI, sometimes automatically generated
Crypto library	C	Library implementing low-level functions
Device driver	C	(If present) interface to hardware (e. g., trusted computing)

2 Benefits

There are several benefits when low-level C code is modeled in the same language as the target application:

- Better integration into the analysis tool, as the tool can fully inspect properties of interest.
- The possibility of combining properties of multiple implementations, giving a stronger specification for verification.
- The possibility of using other analysis technologies, such as symbolic execution, model checking, or fault injection. These technologies are usually not applicable to low-level code.

Model checking for software is specifically useful for concurrent applications, as the outcome of all possible thread and communication schedules cannot be tested effectively. A test run covers one particular scenario [15]. In software where multiple threads [20] of execution work in parallel, a test run executes one particular thread schedule. As the schedule is typically non-deterministic, even repeated test runs cover only a part of all behaviors. Different verification approaches are required for more exhaustive verification. Model checking has the advantage that it is fully automated, but given verification tools for Java require that the entire application exists as Java bytecode [22] or that side-effects of system-specific code are modeled by a special library [4].

Similarly, fault injection tools also require that code is available in a platform that the tool supports [3, 2]. Conversion of so-called checked exceptions from JNI to Java would allow such tools to have a richer view of the library, including exceptions returned from C code.

As complex computations are inevitably simplified when extracting only key properties, the resulting model code would also be more efficient than the original one. This is another benefit both for model checking and other analysis types, because analysis can scale to larger applications.

3 Implementation Strategy

A model of a library function may consist of a *stub,* implementing only a subset of the real functionality [6]. The stub has to be precise enough to allow for execution of a test case of interest. For cryptographic functions and security APIs, certain properties of their behavior help us to write such stubs:

- Cryptographic functions can be replaced with a stub that either returns clear text (for matching keys) or a pseudo cipher text that differs in a simple way. For example, each string may be preceded with a special marker character to mark it as encrypted. This marker is removed upon encryption. Because the goal of software verification is only to ensure that encryption is used whenever necessary, the lack of security of this "encryption scheme" is not a problem.

Java interface:

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

C implementation:

```
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;
    }
...
```

Figure 1: C implementation of a Java native method.

- Security APIs often work in a binary way, either granting or denying access. This can be modeled as a non-deterministic decision.

In both cases, the exact way the C security library works is often irrelevant for testing an application. The library has to implement high-level properties such as providing a secure one-way hash function. Such properties can be analyzed in isolation of the application, for example through cryptanalysis. When analyzing the Java application, only correct usage of the functionality is important.

Therefore, stubs should model preconditions that the Java application must meet when calling the API. Such preconditions can be extracted from assertions in the C implementation. Other properties, such as a correct sequence of calls, may also be accessed by more advanced inspection techniques on the C code, such as program slicing [13].

Figure 1 shows a part of the API for Trusted Computing for Java [12, 21]. In this code, method `nonce_set` is declared to be native in Java, and implemented in C. The Java Native Interface declaration requires the expanded class name of the method and a lengthy signature, but the interesting part is the C implementation of the method. In the C code, the array length of the last argument is checked against a constant that is defined elsewhere. This check is not part of the Java program! However, knowledge of JNI calling conventions allows for a translation of the if-expression from C to Java, where it can be verified even if the C code is subsumed by a stub.

Previous work has implemented a similar mapping for the verification of low-level C libraries [19, 18]. The focus was on generating code annotations, but we aim at generating executable code that does not require extra tool support for analysis. By leveraging tools that represent program structure in XML form, we have a unified representation of the problem [10, 14]. Finally, we hope to include recent advances in reverse engineering to infer properties relating to correct sequences of API calls [8].

References

[1] T. Arnold and L. Van Doom. The IBM PCIXCC: a new cryptographic coprocessor for the IBM eServer. *IBM J. Res. Dev.*, 48(3–4):475–487, 2004.

[2] C. Artho, A. Biere, and S. Honiden. Enforcer – efficient failure injection. In *Proc. Int. Conference on Formal Methods (FM 2006)*, Canada, 2006.

[3] C. Artho, A. Biere, and S. Honiden. Exhaustive testing of exception handlers with enforcer. *Post-proceedings of 5th Int. Symposium on Formal Methods for Components and Objects (FMCO 2006)*, 4709:26–46, 2006.

[4] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Efficient model checking of networked applications. In *Proc. TOOLS EUROPE 2008*, volume 19 of *LNBIP*, pages 22–40, Zurich, Switzerland, 2008. Springer.

[5] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Tools and techniques for model checking networked programs. In *Proc. SNPD 2008*, Phuket, Thailand, 2008. IEEE.

[6] E. Barlas and T. Bultan. Netstub: a framework for verification of distributed Java applications. In *Proc. 22nd Int. Conf. on Automated Software Engineering (ASE 2007)*, pages 24–33, Atlanta, USA, 2007. ACM.

[7] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn. *A practical guide to trusted computing*. IBM Press, 2007.

[8] Vitaly Chipounov and George Candea. Reverse-Engineering Drivers for Safety and Portability. In *4th Workshop on Hot Topics in System Dependability (HotDep)*, San Diego, USA, 2008.

[9] H. Dobbertin. Cryptanalysis of MD4. *J. Cryptology*, 11(4):253–271, 1998.

[10] K. Gondow, T. Suzuki, and H. Kawashima. Binary-level lightweight data integration to develop program understan ding tools for embedded software in C. In *Proc. 11th Asia-Pacific Software Engineering Conference (APSEC 20 04)*, pages 336–345, Washington, USA, 2004. IEEE Computer Society.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[12] Institute for Applied Information Processing and Communications. *Trusted Computing for the Java Platform*, 2009. `http://trustedjava.sourceforge.net/`.

[13] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[14] K. Maruyama and S. Yamamoto. A tool platform using an XML representation of source code informati on. *IEICE – Trans. Inf. Syst.*, E89-D(7):2214–2222, 2006.

[15] D. Peled. *Software Reliability Methods*. Springer, 2001.

[16] Sun Microsystems, Santa Clara, USA. *How to Implement a Provider in the Java Cryptography Architecture*, 2009. `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/HowToImplAProvider.html`.

[17] Sun Microsystems, Santa Clara, USA. *Java Native Interface*, 2009. `http://java.sun.com/javase/6/docs/technotes/guides/jni/`.

[18] G. Tan and J. Croft. An empirical security study of the native code in the jdk. In *Proc. 17th Conf. on Security Symposium (SS '08)*, pages 365–377, San Jose, USA, 2008. USENIX Association.

[19] G. Tan and G. Morrisett. ILEA: inter-language analysis across Java and C. In *Proc. 22nd annual ACM SIGPLAN Conf. on Object-oriented programming systems and applications (OOSPLA 2007)*, pages 39–56, Montreal, Canada, 2007. ACM.

[20] A. Tanenbaum. *Modern operating systems*. Prentice-Hall, 1992.

[21] T. Vejda, R. Toegl, M. Pirker, and T. Winkler. Towards trust services for language-based virtual machines for grid computing. In *Proc. 1st Intl. Conf. on Trusted Computing and Trust in Information Technologies (Trust '08)*, pages 48–59, Villach, Austria, 2008. Springer.

[22] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.