

Bandwidth Efficient Tamper Detection for Distributed Java Systems*

Mike Jochen
University of Delaware
jochen@cis.udel.edu

Lisa Marvel
U.S. Army Research Laboratory
marvel@arl.army.mil

Lori L. Pollock
University of Delaware
pollock@cis.udel.edu

Abstract

The benefits of distributed computation present complex security considerations beyond those associated with the traditional computing paradigm. This paper describes a bandwidth efficient approach to authenticate distributed Java code. Our system utilizes steganographic techniques to embed a cryptographic checksum as a tamper detection mark into Java class files. The properties of this mark make our system desirable in applications where low bandwidth utilization is a requirement (e.g., wireless networks and low power devices). We have implemented our system in Java and evaluated its performance through empirical study. Analysis indicates that our system detects any degree of alteration to a marked Java class file and can do so within a reasonable amount of time.

1 Introduction

The increasing popularity of distributed computation motivates a heightened awareness of the security issues associated with executing code from a possibly unknown and untrusted origin, or code that may have been altered during transit to the local host. Consequently, a distributed code user should be aware of the source, correctness, intentions, and integrity of the code in execution on their local machine. From a security standpoint, this knowledge relates to the code's well-formedness (e.g., syntax and type compatibility), behavior (e.g., memory access and resource utilization), and integrity (e.g., Did the code arrive from a trusted host, intact and unchanged?). We propose utilizing a hybrid steganographic-cryptographic approach to embed a

Tamper Detection Mark (TDM) into the distributed (mobile) code¹ as a way to address the issues of code integrity and author authentication.

There are many active areas of research in security for distributed (mobile) codes [14, 20, 23]. Much has been done to protect the local host from poorly or maliciously written code that attempts to take advantage of improper memory accesses (e.g., stack overflows, pointer abuse, and unsafe type manipulations) [18, 13]. This paper focuses on the question: "How do we determine in a bandwidth efficient manner, if a distributed code has been substituted or modified in some form from its original state?" Validating a remote code for execution on a local host can prevent undesirable situations where, for example, a malicious code tricks a local user into revealing or destroying important private information. This validation assumes that trust has been established in the server of the remote code. Existing methods to validate mobile and distributed code include using digital certificates, encrypting the code, and using digital signatures to sign applications [7, 12]. Each of the above techniques has shortcomings which motivate continued research in new approaches to provide for a more efficient delivery of code authentication data. The TDM provides a bandwidth efficient method to validate mobile code without the drawbacks of digital signature techniques currently in use. The properties of the TDM make it especially attractive for use in applications for low power, mobile devices, and wireless networks.

Steganography is the process of hiding information in other information [9]. Steganography has been used throughout the ages as a means of subliminal communication. A secret message is embedded in seemingly innocuous cover data (e.g., a picture, text, or symbols). The embedding process usually exploits statistical randomness or noise within the data of the cover medium

*Prepared through collaborative participation in the Communications and Networks Consortium sponsored by the U. S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0011. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

¹In this paper, code being transmitted in a distributed system over a network is considered to be mobile code with all the security concerns of other mobile codes, where mobile code is defined to be any code that has been compiled on a remote machine other than the local one in which it is executing. For this reason, we sometimes use the terms distributed code and mobile code interchangeably.

to hide the secret. The secret message can subsequently be delivered unnoticed because its existence is difficult to detect. Hence, this technique permits the exchange of information to occur without there even being the perception that communication ever took place.

In this paper, we present a new framework to enable users of mobile Java code in a distributed system to authenticate code and certify that the code has arrived unaltered from a trusted host. Using steganography to communicate encrypted authentication data has several advantages over alternative approaches. Embedding the authentication data within the code (1) reduces bandwidth requirements, (2) reduces the chance of separating authentication data from the code, and (3) obscures the existence of the authentication data from casual view. Because the marked code is semantically equivalent to its original, unmarked form, our framework gives the user the option of whether to validate a code or not before execution begins; our technique permits code containing a TDM to be executed without any additional run-time or pre-run-time processing, if desired. Our system can be utilized as a stand alone system or as an API for inclusion in a distributed computation environment.

The remainder of the paper is organized as follows. Section 2 provides relevant background on current tamper detection techniques for distributed and mobile code. Section 3 briefly describes the Java environment. In Section 4, we present our technique for hiding and extracting information within a Java class file. Section 5 evaluates our TDM technique through empirical study. In Section 6, we present related work. Section 7 summarizes and discusses our future research directions.

2 Current Integrity Validation Methods

Current methods to validate the integrity of mobile code include signing the code with a digital signature and certifying the signature with a digital certificate.

Signed code (e.g., signed Java archives known as JAR files [12]) can address the issue of identifying compromised mobile code by appending a digital signature to the code in some fashion. The main shortfalls of signatures are: (1) the signature requires extra space and bandwidth, and (2) the signature can become separated from the code. A unique denial of service attack can be orchestrated by simply removing a digital signature from the signed code during transit. Once the code arrives at the destination, authentication will not be possible. The destination has no way of knowing whether the absence of the signature is due to network error or malicious intent. Removal or alteration of the more bandwidth efficient TDM from code mandates tampering with the code. The TDM system immediately alerts

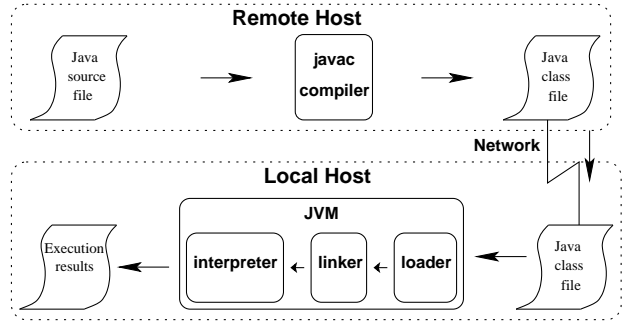


Figure 1. Typical steps in compiling, distributing, and executing a Java program.

the destination of tampering in such instances.

A certificate permits one to validate the identity of the author of mobile code which has been downloaded to the local host [7]. Certificates are lacking in several areas: (1) the certificate can be separated from the code, (2) extra time, bandwidth, and resources must be used to handle the certificate, (3) the degree of confidence one can place in a certificate is directly impacted by the integrity of the issuing authority, (4) a certificate can not inform the user if a mobile code was modified in any way between the time it was compiled on the trusted host and the time it is run on the local machine, and (5) the certificate server provides a single point of vulnerability; once the certificate server has been compromised, the entire system is vulnerable.

3 Java Distributed Computation

Java has significantly popularized mobile distributed computation [4, 1], increasing both its use and existence through Java's write-once-run-anywhere nature. The process of generating and executing mobile Java code is depicted in Figure 1. In the first phase, the remote host compiles the Java source code into Java class file(s). The program is then distributed to local hosts which run the code on a Java Virtual Machine (JVM). The JVM handles loading, verifying, linking, and interpreting Java class files.

The structure of a Java class file is defined in [12]. An important component of the class file is the constant pool table. The constant pool table is an array of constant objects which store information relating to class names, method names, and method type signatures, among other constants. This table is used extensively during the loading and linking phases at run-time to verify the correct form and run-time behavior of the class. It is also used during run-time for memory allocation and method invocation. There are many references

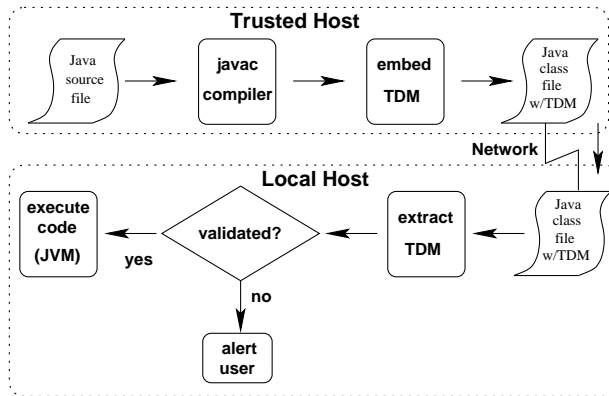


Figure 2. Framework for the TDM system.

to entries in the constant pool table throughout the class file.

4 Tamper Detection in a Class File

Our overall framework for tamper detection in a Java class file is depicted in Figure 2. After Java compilation to bytecode is complete on the remote host side, we generate and embed the TDM into the executable image of the Java code using steganography. The TDM is a cryptographic checksum of the class file. To generate the TDM for a given class file, the remote host computes a hash value of the entire class file. This hash value is then encrypted, creating a Digital Authentication Code (DAC), and subsequently embedded within the Java code. Within this framework, the remote host is now viewed as a trusted host which produces the mobile code with the embedded TDM. Our system is an open one in that the rules for embedding, extracting, and validating the TDM are open and available to the public. The security of our system comes from encrypting the hash value to protect the TDM.

The challenge arises in trying to find where in the class file the TDM can be embedded. The requirement is a segment that contains noise or ambiguity (i.e., gives the appearance of some degree of randomness) such that the segment can be manipulated to encode the TDM, but also maintain the semantic meaning of the original code. In addition, an embedding process that does not increase bandwidth requirements is desirable, making the approach of adding annotations to the code a non-viable option. The ordering of the constant pool table fits these requirements. The original ordering of the constant pool table appears arbitrary and can be manipulated to represent the TDM while keeping the resulting class file semantically equivalent to the original class file.

Algorithm 1 Algorithm for embedding a TDM in a Java class file at the trusted host.

Input: Java_classfile, C; encryption_key, Key;

Output: Java_classfile, C, with embedded TDM;

```

1: sort(C.ConstantPoolTable);
2: foreach object o in classfile C do
3:   if o refers to a constant pool object in C then
4:     update reference in o to new object location;
5:   end if
6: endfor
7: computed_TDM ← hash(C);
8: DAC ← encrypt(computed_TDM, Key);
9: permute(C.ConstantPoolTable, DAC);
10: foreach object o in classfile C do
11:   if o refers to a constant object in C then
12:     update reference in o to new object location;
13:   end if
14: endfor
15: RETURN C;

```

4.1 Embedding the TDM

Algorithm 2 Algorithm for extracting a TDM in a Java class file at the local host.

Input: Java_classfile, C; encryption_key, Key;

Output: Boolean (T/F) if C is validated;

```

1: sort(C.ConstantPoolTable);
2: foreach object o in classfile C do
3:   if o refers to a constant pool object in C then
4:     update reference in o to new object location;
5:   end if
6: endfor
7: computed_TDM ← hash(C);
8: DAC ← permuteInverse(C); /* Extract DAC */
9: extracted_TDM ← decrypt(DAC, Key);
10: RETURN (computed_TDM = extracted_TDM);

```

While manipulating the order of the constant pool table to encode a TDM may appear to be a straightforward approach, several issues must be addressed to maintain correctness and prevent an increase in bandwidth requirements. (1) There is no standard for the order of the constant pool table. The same Java source code compiled by two different compilers can result in different orderings of the constant pool table entries. (2) There exist numerous references to the constant pool table throughout the entire class file. Any change to the order of the constant pool table in an existing class file requires that all references be updated to reflect this new order. (3) The hiding capacity of the constant pool table is directly related to the size of the table. A table with many entries is able to hold more hidden data than a table with few entries. (4) Naive reordering can increase bandwidth requirements due to the addressing schemes used by the JVM for constant pool table references, thus reordering schemes must be sensitive to these addressing schemes. The algorithm for embedding the TDM is

```

public void <init>()
0:  aload_0
1:  invokespecial    java.lang.Object.<init>()V (1)
4:  return
public static void main(String[])
0:  getstatic        java.lang.System.out Ljava/io/PrintStream; (2)
3:  ldc              "Hello World!" (3)
5:  invokevirtual    java.io.PrintStream.println (Ljava/lang/String;)V (4)
8:  return

```

(a) Method table for original “Hello World!”

```

public void <init>()
0:  aload_0
1:  invokespecial    java.lang.Object.<init>()V (8)
4:  return
public static void main(String[])
0:  getstatic        java.lang.System.out Ljava/io/PrintStream; (9)
3:  ldc              "Hello World!" (10)
5:  invokevirtual    java.io.PrintStream.println (Ljava/lang/String;)V (6)
8:  return

```

(b) Method table for “Hello World!” with TDM

Figure 3. Original and processed method tables for “Hello World!”

given in Algorithm 1.

Given these four issues, our approach to embedding the TDM starts with sorting the constant pool table to establish a common base state. The sorting step addresses issue (1), and insures that the local host and the trusted host are starting with a constant pool table in an identical canonical form. The reordering of the constant pool necessitates that all references to constant pool objects be updated throughout the entire class file to reflect the new location of the constant objects to which they refer. The class file is said to be in canonical form after sorting and updating all references to the constant pool table.

A TDM of the class file in canonical form is computed and encrypted, producing the DAC. This DAC is represented by permuting the constant pool table accordingly. The encoding capacity of the constant pool table guides the choice of hash and encryption algorithms (i.e., a small constant pool table can only handle hash algorithms that yield smaller hash values). Thus, the hash and encryption algorithm choice addresses issue (3). We discuss more on this issue in Section 5. The reordering algorithm also addresses issue (4) through careful consideration of the addressing schemes utilized by the JVM and the impact of reordering on bandwidth requirements with respect to these addressing schemes.

To reorder the constant pool table, we select the n^{th} permutation of the table via the permutation algorithm given in [10], where n represents the value of the DAC. The permutation algorithm utilized by the producer to embed the TDM is the same one utilized by the consumer to extract the mark. After the constant pool has been permuted, the new ordering of the constant pool is used to re-index all references into the original constant pool table once more to reflect the final ordering of the table. This updating process addresses issue (2) mentioned above regarding the numerous references to the constant pool table throughout the class file. Thus, the class file’s references to constant pool table objects are updated twice, once after sorting to put the class file in canonical form to compute the hash, and a second time to reflect the final order of the constant pool table. This

re-indexed class file can then replace the original class file as the code to be distributed to the local host.

4.2 Extraction and Validation

On the local host side, the TDM is used to validate the code. The algorithm performed at the local host is presented in Algorithm 2. To extract the TDM and validate a class file, the local host replicates some of the steps performed by the trusted host that embedded the mark. The local host must first transform the class file to canonical form. The hash, computed_TDM, of the class file in canonical form is computed. The ordering of the constant pool table of the class file as received from the trusted host is compared to the ordering of the constant pool in canonical form. This comparison is used to determine which of the n permutations is represented by the constant pool table as received from the trusted host and yields the DAC, which can be decrypted to provide the extracted_TDM. The extracted_TDM is compared with the local computed_TDM. If computed_TDM equals extracted_TDM, then the class file is validated and the local host can safely proceed with execution. If these values do not agree, then the class file was altered in some manner between insertion of the mark and validation or an incorrect key was used to generate the DAC, indicating origination from an untrusted host. The local host can decide not to execute the code or can proceed with execution at its own risk.

As an example, Figures 3 and 4 show the method tables and constant pool tables for the “Hello World!” class file before and after embedding the TDM. The class file with the embedded TDM is the class file that the local host would receive from the trusted host. Note the new order of the constant pool in Figure 4(b) as compared to the original order in Figure 4(a). Note further that entries within the constant pool which refer to other entries within the constant pool are re-indexed in Figure 4(b). There exists only a slight difference between the method tables in Figures 3(a) and 3(b). The JVM bytecode opcodes stay the same, only the operand ad-

dresses (i.e., the values in parenthesis at the end of a line) change where there are references to constant pool objects. These references are also re-indexed to reflect the new position of constant objects within the constant pool table. Because class files can be produced by different compilers, generated by hand, or processed after compilation by a code obfuscator or optimizer, the constant pool tables for the same Java code can vary greatly, thus obscuring the existence of the TDM in a permuted constant pool.

The embedding of the TDM occurs just after compilation on the trusted host. This process could be integrated with the Java compiler or maintained as a separate step. Validation of the class file occurs just before invocation of the JVM. This step could be integrated with the class loader and become part of the JVM or it could also be maintained as a separate step.

5 Evaluation of the TDM Technique

In this section, we present analysis of our approach in terms of the security of the TDM and the performance of the processes of embedding and extracting the TDM.

5.1 Security Properties

Our system will always be able to extract a mark from a legal class file, as there is always an order to the constant pool table which enumerates one of the possible permutations for those table entries. The extracted mark from this table will either be a valid TDM or garbage.

A valid TDM will provide the hash value of the class file. The ability to decrypt the DAC and match the resulting value with the local computed_TDM verifies that the code was sent unaltered by a trusted host. If the TDM is not valid, then the class file has been altered from its protected state or the constant pool table has been reordered, thus destroying the mark (e.g., the order of the constant pool table may have been scrambled by recompilation yielding an invalid mark). In some sense, our system could yield a false alarm in terms of identifying a class file as invalid. Our system will indicate that the class file has been altered should the order of the constant pool table be changed, even if the semantics of the class file remain equivalent to its original form. To the lay person, this may seem to be equivalent to the legend of the “boy who cried wolf” – the class file will still perform exactly as intended, nothing seems to really be wrong. The fact remains in this instance that somehow, this class file has been altered from its original state. The only way to alter or overwrite the original TDM of a class file without changing the semantics of the code is to physically alter the order of the constant pool table of the respective

class file. Clearly, this process involves tampering with the class file itself. Thus, our system alerts the user to any change to the class file, no matter how insignificant. This flagging action permits the user to take other actions to determine the source of the tampering and possibly prevent such tampering in the future.

The security of our system is reliant upon the security of the encryption and hash functions used to create the TDM. As long as the hash function employed by our system has a low probability of collision, the probability of an opponent finding another class file that has the same hash value as the original class file is low. Compounding the difficulty of a collision attack is the fact that the class file that collides with the code under attack must be a legal Java class file, or the JVM will signal an exception. When using a good hash function, a 1-bit change to the class file results in a change in 50% of the bits in the hash value. Thus, we chose two well-known hash functions, MD5 and where possible, SHA-1. Both of these algorithms are accepted in many publicly available digital signature and encryption software packages, though SHA-1 is considered to be the stronger of the two algorithms [21].

Since our system is an open one, where the procedures to embed, extract and validate the TDM are public, we protect the hash of the class file by encrypting the hash value. This ties the security of the system to the choice of encryption algorithm, the size of the key for the encryption, and the security of key distribution and key management. While any secure block cipher algorithm will suffice, our initial implementation utilizes DES[15]. While there has been argument over key length and s-box selection for DES, the ready availability of the coded algorithm in Java made it a natural choice for our proof-of-concept system. The final choice of encryption algorithm is left to the user. Use of DES means that the hash value is encrypted in 64 bit blocks. MD5 is a 128 bit hash, thus its hash value can be encrypted in two blocks. SHA-1 is a 160 bit hash which must be padded for encryption. The SHA-1 hash value is padded with 32 bits to reach a 192 bit value which can be encrypted in three 64 bit blocks.

Our system assumes that the secret key for the DAC is communicated over a secure channel at any time before validation of the code is to occur. While key distribution and management is critical to security interests, this work does not attempt to address this problem. Key distribution and management is an area of active research with several worthwhile treatments and analyses. A good starting point of information for the reader can be found in [22, 3].

Reverse engineering of a TDM protected class file by a malicious host and subsequent recompilation will re-

```

1)CONSTANT_Methodref[10](class_index = 6, name_and_type_index = 15)
2)CONSTANT_Fieldref[9](class_index = 16, name_and_type_index = 17)
3)CONSTANT_String[8](string_index = 18)
4)CONSTANT_Methodref[10](class_index = 19, name_and_type_index = 20)
5)CONSTANT_Class[7](name_index = 21)
6)CONSTANT_Class[7](name_index = 22)
7)CONSTANT_Utf8[1](<init>)
8)CONSTANT_Utf8[1](<V>)
9)CONSTANT_Utf8[1](<Code>)
10)CONSTANT_Utf8[1](<LineNumberTable>)
11)CONSTANT_Utf8[1](<main>)
12)CONSTANT_Utf8[1](<(Ljava/lang/String;)V>)
13)CONSTANT_Utf8[1](<SourceFile>)
14)CONSTANT_Utf8[1](<Hello.java>)
15)CONSTANT_NameAndType[12](name_index = 7, signature_index = 8)
16)CONSTANT_Class[7](name_index = 23)
17)CONSTANT_NameAndType[12](name_index = 24, signature_index = 25)
18)CONSTANT_Utf8[1](<Hello World!>)
19)CONSTANT_Class[7](name_index = 26)
20)CONSTANT_NameAndType[12](name_index = 27, signature_index = 28)
21)CONSTANT_Utf8[1](<Hello>)
22)CONSTANT_Utf8[1](<java/lang/Object>)
23)CONSTANT_Utf8[1](<java/lang/System>)
24)CONSTANT_Utf8[1](<out>)
25)CONSTANT_Utf8[1](<Ljava/io/PrintStream;>)
26)CONSTANT_Utf8[1](<java/io/PrintStream>)
27)CONSTANT_Utf8[1](<println>)
28)CONSTANT_Utf8[1](<(Ljava/lang/String;)V>)

```

(a) Constant pool table for original “Hello World!”

```

1)CONSTANT_NameAndType[12](name_index = 7, signature_index = 26)
2)CONSTANT_NameAndType[12](name_index = 22, signature_index = 27)
3)CONSTANT_Class[7](name_index = 15)
4)CONSTANT_Utf8[1](<Code>)
5)CONSTANT_Utf8[1](<(Ljava/lang/String;)V>)
6)CONSTANT_Methodref[10](class_index = 16, name_and_type_index = 2)
7)CONSTANT_Utf8[1](<init>)
8)CONSTANT_Methodref[10](class_index = 3, name_and_type_index = 1)
9)CONSTANT_Fieldref[9](class_index = 14, name_and_type_index = 20)
10)CONSTANT_String[8](string_index = 21)
11)CONSTANT_Utf8[1](<out>)
12)CONSTANT_Utf8[1](<LineNumberTable>)
13)CONSTANT_Utf8[1](<Ljava/io/PrintStream;>)
14)CONSTANT_Class[7](name_index = 23)
15)CONSTANT_Utf8[1](<java/lang/Object>)
16)CONSTANT_Class[7](name_index = 19)
17)CONSTANT_Utf8[1](<Hello.java>)
18)CONSTANT_Utf8[1](<SourceFile>)
19)CONSTANT_Utf8[1](<java/io/PrintStream>)
20)CONSTANT_NameAndType[12](name_index = 11, signature_index = 13)
21)CONSTANT_Utf8[1](<Hello World!>)
22)CONSTANT_Utf8[1](<println>)
23)CONSTANT_Utf8[1](<java/lang/System>)
24)CONSTANT_Utf8[1](<Hello>)
25)CONSTANT_Class[7](name_index = 24)
26)CONSTANT_Utf8[1](<V>)
27)CONSTANT_Utf8[1](<(Ljava/lang/String;)V>)
28)CONSTANT_Utf8[1](<main>)

```

(b) Constant pool table for “Hello World!” with TDM

Figure 4. Original and processed constant pool tables for “Hello World!”

sult in a class file with an invalid TDM. For this sort of an attack to succeed on a TDM protected class file, the malicious host must have the secret key. Without the secret key, the TDM in the compromised class file will not match the locally generated authentication data by the local host. The local host will immediately be able to detect that the class file under consideration has been altered.

Defense against versioning attacks can be handled by updating the secret key used to generate and validate TDMs. For each new version of a mobile code, a new key can be used to generate a new TDM.

We have designed our system to be modular in that the encryption and hash algorithms can be changed without great difficulty. Thus, our system can readily accommodate other encryption algorithms and hash functions, permitting the system to be updated or adapted to suit personal preference with ease. Of course, each host and client of the system must then utilize the same hash and encryption algorithms.

5.2 Analysis of Time and Space

Potential concerns about performance of a tamper detection technique are (1) that the embed and validate procedures in the approach might take too much time to complete, (2) that the space during embedding and validation is too large to be scalable, and (3) that bandwidth requirements to communicate the authentication data are

increased.

The time to process a given class file depends on two factors, (1) the size of that class file’s constant pool table, and (2) the physical size of that class file plus the size of any interfaces that class file implements. Here, the size of the constant pool table is represented by the variable n . Clearly, sorting the constant pool table (Sort CP) is bounded by $O(n \log n)$. The size of the class file is represented by the variable b_0 . The time to re-index all references to constant pool objects in a given class file of size b_0 is bounded by $O(b_0)$. The subscript j represents the number of interfaces implemented by a class. If a class implements an interface, the implemented interface is included in that class file’s hash value, thus the time to compute the hash value for that class file is the time to hash b_0 plus the j interfaces ($b_1..b_j$), where $\text{hash}(x)$ is the time to compute the hash value for a file of size x . The size of the hash value is h thus, the time to encrypt the hash value is bounded by $O(\text{encrypt}(h))$, where $\text{encrypt}(x)$ is the time to encrypt x . The permutation algorithm takes time $O(n)$ while finding the permutation takes time $O(h^3)$. The permutation takes time $O(n)$ because the permutation algorithm is linear in n and the entire constant pool is permuted so that the entire pool order looks random. Finding the permutation takes time $O(h^3)$ because the inverse permutation algorithm is cubic in n and the TDM is represented in part of the permutation just large enough to encode the encrypted hash value, not the entire constant pool table.

In summary, the time to embed a TDM is bounded by $O(n \log n + \text{hash}(b_0 \dots b_j) + \text{encrypt}(h))$. The time to extract a TDM is bounded by $O(h^3 + \text{hash}(b_0 \dots b_j) + \text{decrypt}(h))$. The critical factors are the size of the class file (and any implemented interfaces), and the size of the constant pool table. For class files with very large constant pool tables, the size of the constant pool table becomes the dictating factor in the time analysis. For large class files with smaller constant pool tables (e.g., a class file with many large methods), the file size of the class file drives the time analysis.

Space utilization during embedding and validating the TDM for a given class file is linear in the size of the class file. We manipulate one copy of the class file. The physical size of the class file after embedding the TDM almost always remains the same. Thus, bandwidth requirements are not increased in general, and in some cases are actually decreased. In one case, the bandwidth requirements increased slightly.

5.3 Implementation and Empirical Study

Our system was implemented entirely in Java, utilizing the BCEL [6] and Cryptix [5] packages. The goal of our empirical study was to investigate the amount of time and space our technique requires to process a set of Java codes. After timing data was collected, processed benchmarks were executed to verify program correctness. All experiments were performed on a 300MHz Sun Ultra-10 machine with 192MB physical memory running the Java 2 platform. We tested our implementation on the SpecJVM98 suite where reported times are the medians of three execution times.

The TDM system computes an MD5 [19] or SHA-1 [16] hash of the class file based on the hiding capacity of the constant pool table (given issue (3) above). The resulting 128 or 160 bit hash value (from MD5 or SHA-1, respectively) was encrypted before being embedded within the code. To embed a 128 bit number, the constant pool must have a minimum size of 35 entries. For 35 lines, there are $35!$ possible permutations of the constant pool table entries. This number of re-orderings is sufficient to represent a 128 bit value. This requirement can be understood by examining the following relation: $34! < 2^{128} < 35!$. The 160 bit SHA-1 hash requires a constant pool size of at least 41 entries (47 entries after padding).

We report results for seven of the ten SpecJVM98 benchmarks. Three of the benchmarks (**jack**, **javac**, and **mpegaudio**) contained class files with constant pool tables that did not meet the minimum size requirement. The smallest possible class file (an empty class) has a constant pool table size of 13 entries. The smallest inter-

face (an empty interface) has a constant pool table size of 7. In our system, interfaces do not contain TDMs. The TDM for each interface is included in the TDM for every class that implements that interface. Thus, we do not need to worry about encoding a TDM in a constant pool table with only 7 entries. We are currently addressing the problem of dealing with class files whose constant pool tables do not meet the minimum requirement. The smallest constant pool our technique can currently handle is 21 entries. For constant pools smaller than 35 entries, half of the 128 bits for the MD5 hash are used to construct the DAC. As a point of comparison, the very simple “Hello World!” code has a constant pool with 29 entries. Including implementations of hash and encryption algorithms that generate smaller DACs will allow our system to process class files with constant pool tables that do not meet the current minimum size requirement. A naive solution would be to simply pad the constant pool table with enough entries to encode the desired TDM. However, doing so would violate steganographic techniques as we would then be adding information to the system, not encoding information within the available data.

All benchmarks with embedded TDMs executed correctly and without any change in performance. Benchmarks with TDMs were also altered to test the ability of our system to detect various degrees of tampering. The alterations for each class file ranged from editing a single bit with a hexadecimal code editor to reverse engineering and recompilation. As was expected, any alteration resulted in a different TDM and thus, was detected by our system.

5.4 Performance Results

Table 1 shows timing data for the benchmarks that were processed by our system. Most benchmarks contained more than one class file. Average class file sizes are given (in KB) as well as the total size of all class files for each benchmark. Table 1 also lists the average and total number of entries within the constant pool tables of the class files for each benchmark. The approximate time to transmit the benchmark over a 53 Kbps bandwidth line (approximate bandwidth of a consumer modem) is given in seconds to give the reader some base for viewing TDM system performance. The last four columns indicate the average and total times (in seconds) for the embed and validate processes. Embed time includes time to create and embed the TDM. Validate time includes time to extract and validate the TDM. The average time recorded is the average time taken to process each class file in the given benchmark. We note that our system does not take much time to process each in-

Table 1. Embed and extract times for SpecJVM benchmarks.

| Name | # of Files | File Size (KB) | | Pool Size (# of entries) | | Xmit Time (seconds) | Embed Time (seconds) | | Validate Time (seconds) | |
|----------|------------|----------------|-------|--------------------------|-------|---------------------|----------------------|-------|-------------------------|-------|
| | | Avg | Total | Avg | Total | Approx | Avg | Total | Avg | Total |
| mtrt | 1 | 0.84 | 0.84 | 51 | 51 | 0.13 | 0.45 | 0.45 | 0.38 | 0.38 |
| checkit | 3 | 1.92 | 5.76 | 104 | 312 | 0.87 | 0.25 | 0.75 | 0.19 | 0.58 |
| db | 3 | 3.31 | 9.92 | 192 | 577 | 1.50 | 0.37 | 1.1 | 0.22 | 0.67 |
| jess | 5 | 2.08 | 10.4 | 98 | 488 | 1.57 | 0.22 | 1.09 | 0.14 | 0.72 |
| compress | 12 | 1.45 | 17.4 | 79 | 950 | 2.62 | 0.09 | 1.11 | 0.07 | 0.84 |
| check | 17 | 2.32 | 39.43 | 116 | 1965 | 5.95 | 0.11 | 1.94 | 0.07 | 1.16 |
| raytrace | 25 | 2.23 | 55.66 | 86 | 2155 | 8.40 | 0.1 | 2.54 | 0.05 | 1.33 |

dividual class file. It took an average of 0.14 seconds to embed the TDM and 0.09 seconds to extract and validate the TDM per class file.

Figure 5 shows the average embed and validate times per class file for each benchmark. Figure 6 shows the total embed and validate times for each benchmark. The times indicated in both charts are given in seconds. Also included in Figure 6 is the approximate time it would take to transmit each benchmark across a 53 Kbps line. The benchmarks are listed on the x-axis of both charts in order of increasing total constant pool table size. This ordering also corresponds with an increase in the total number of class files per benchmark. As the total size of the benchmark gets larger, the time to process the file increases as expected.

From Figure 5, we observed no consistent correlation between embed (validate) times and class file size or constant pool size. This confirms that there are a number of factors affecting embed (validate) times. Some of these factors include number of interfaces implemented by the class, the number of references in the class file to the constant pool table, and the hash algorithm used. For all benchmarks, the validate time is consistently less than the embed time, something which is desirable for the local user running a distributed code.

Figure 6 indicates how the embed (validate) time increased as the total size of the benchmark increased in our experiments. Again, the total embed time was consistently larger for the total validate time for all benchmarks in our experiments. Lastly, the transmit time over a 53 Kbps line was significantly higher than processing time for larger benchmarks.

Our goal is to communicate the authentication data without requiring additional bandwidth. In our experiments, the size of all of our benchmarks remained unchanged or actually decreased in size, except **check** (2.32 KB file size), which increased by 111 bytes with the TDM. We hypothesize the following reason for this observation. During execution, the JVM uses two different address sizes to load constants from the constant pool table. If the constant to be loaded is in one of the first 255 positions, a one byte address is used, otherwise

a two byte address is used. Thus, rearranging the order of the constant pool table can have an effect on the size of the byte code. If commonly referenced constants are moved into any of the first 255 positions, then the size of the bytecode and the class file will shrink. Conversely, the size of the class file can grow by moving these constants into positions higher than the 255th position. We have implemented the sorting and permutation phases of the process in such a way as to attempt to minimize the size of the TDM protected class file by moving frequently referenced constants to positions with smaller addresses. Some class files in the benchmark experienced a decrease in file size due to these efforts. While we are currently investigating the cause for the slight increase in size of the **check** benchmark, we stress that we are not adding anything to or removing anything from the class file, simply rearranging the order of the constant pool table and updating the corresponding indices.

A change in the size of the class file will not alert anyone to the existence of the TDM for three reasons: (1) the size of the class file carrying a TDM is the exact size that class file should be given that ordering of the constant pool table, (2) without access to the original source code and knowledge of how the class file was compiled or created, it is impossible to tell what the original size of that class file should be, and (3) there is no specification dictating a certain order to the constant pool table thus, any ordering is legal.

6 Related Work

There are many facets of security as applied to distributed and mobile codes [23, 13]. The first considerations are whether to protect the host from the code or to protect the code from the host. Our work addresses the issues of code integrity and authentication. A vast amount of work addresses other security issues related to foreign code execution on local and remote machines. In Java, the strong typing and rigorous run-time checks performed by the JVM protect the local machine from poorly formed Java code according to a predefined se-

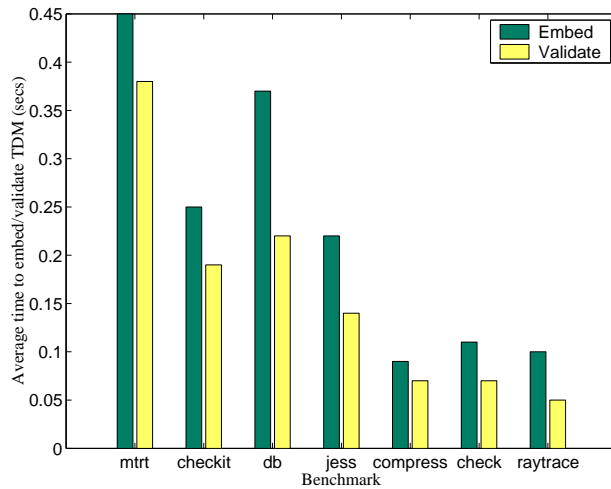


Figure 5. Average embed and extract time (in secs) per class file per benchmark.

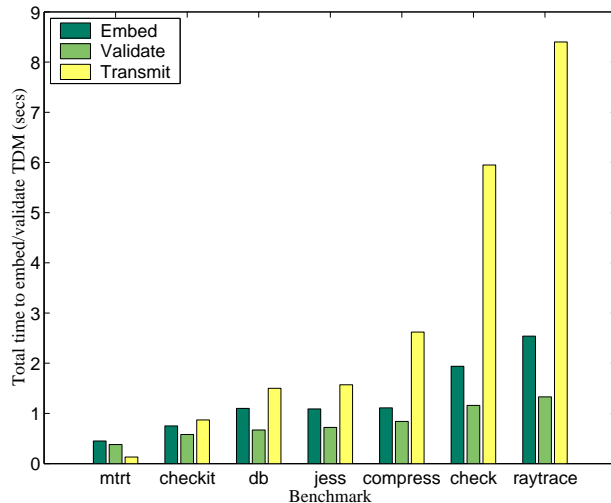


Figure 6. Total embed and extract times (in secs) per benchmark.

curity policy [12]. For example, these checks prevent against improper memory access and assignment.

Kozen [11] and Necula [17] have extended protection to the local machine via certified code and proof-carrying code, respectively. Both of these techniques provide a means to guarantee the correct run-time behavior of a code in terms of some predefined security policy. Certified code is produced by a certifying compiler and is guaranteed in terms of type safety, control flow safety, and memory safety. Proof-carrying code can enforce any predefined security policy via a compiler generated, user-defined proof. Complete automation of proof generation for more involved security policies is still an area of active research. Appel and Felten [2] have extended the proof-carrying code framework to the idea of authentication. This distributed authentication framework provides for the authentication of requests via user generated proofs.

Executing code in a restricted memory area is known as utilizing a reference monitor or sandbox [12]. The sandbox paradigm is the security model used by the JVM when running Java applets and many commercially available mobile code security solutions. Access to system resources (e.g., local file system) can be restricted to prevent malicious code from performing undesirable actions while in execution on the local machine. While the sandbox can offer some degree of protection from malicious code, its presence can be rather cumbersome in situations with the complexity of managing security policies for certain applets, hosts, and domains bogging down the user. Further, the sandbox does not address the issues of authentication of the author nor the validation of code integrity. Hauswirth et al. have extended the JVM's security mechanisms while simplifying its implementation via the Java Secure Execution Framework (JSEF) [8]. The JSEF streamlines Java's security policy model while providing greater functionality through a more expressive, hierarchical security framework. While the JSEF vastly improves on the standard security model provided by the Java architecture, the issues of authentication and validation are again left to other means.

Encrypted code can be used to validate the author of the code and to protect the code in transit. Here, the author encrypts the entire executable image, which is decrypted before execution. Assuming that the user has the correct decryption key, if there are any problems decrypting the code, the user knows that either the code was altered during transit, or that the code did not come from the trusted host (i.e., the wrong key was used to encrypt the code). We do not consider encrypted code to be a viable option because encrypted code requires special steps (i.e., decryption) before the user can execute

the code. We submit that the user should have the option of whether or not to validate the code and whether or not to run the code based on the results of validation if it is performed. Optional validation provides security risk management to clients of mobile code.

7 Conclusions and Future Work

We have designed and implemented a bandwidth efficient system to enable a user to validate the integrity of distributed Java code before execution of that code begins. Validation assures the user that the code was generated by a trusted host and that the code was not altered in any way between the time the code was marked and the time the user performs the validation process. We utilize steganographic techniques to embed a cryptographic checksum as authentication data within the code, thus eliminating the risk of separation and the extra bandwidth requirements of conventional digital signatures and certificates. Our system does not suffer from the vulnerabilities of a certificate server with its single point of failure. Validation is optional with our system, providing security risk management and making it less obtrusive than encrypted code, which mandates decryption before execution can begin.

We have shown that our system works well as a stand alone unit, but it may be incorporated within a Java compiler and a distributed framework for automated, behind-the-scenes validation of distributed code. Analysis indicates that the system detects any degree of tampering with a compiled Java class file and can do so within a reasonable amount of time. This makes our system palatable to the potentially impatient user.

While Java is the most popular means of writing and providing distributed code over open networks of heterogeneous machines, we are currently looking into handling distributed codes written in other languages.

“The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U. S. Government.”

References

- [1] W. Amme, N. Dalton, and J. von Ronne. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI-01)*, 2001.
- [2] A. W. Appel and E. W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proceedings of the ACM 6th Conference on Computer and Communications Security (CCS-99)*, pages 52–62, Nov. 1999.
- [3] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In B. Pfitzmann, editor, *Advances in Cryptology – EURO-CRYPT*, pages 451–472. Springer-Verlag, 2001.
- [4] D. M. Chess. Security issues in mobile code systems. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 1998.
- [5] Cryptix Foundation Limited. Cryptix library. <http://www.cryptix.org/>.
- [6] M. Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Freie Universität, 1998.
- [7] A. K. Ghosh. On certifying mobile code for secure applications. In *International Symposium on Software Reliability Engineering*, page 381. IEEE, 1998.
- [8] M. Hauswirth, C. Kerer, and R. Kurmanowitsch. A secure execution framework for Java. In S. Jajodia and P. Samarati, editors, *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS-00)*, pages 43–52, Nov. 2000.
- [9] S. Katzenbeisser and F. A. P. Petitcolas. *Information hiding techniques for steganography and digital watermarking*. Artech House, 2000.
- [10] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, 1981. 2nd ed.
- [11] D. Kozen. Efficient code certification. Technical Report 98-1661, CS Dept, Cornell University, 1998.
- [12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
- [13] G. McGraw and E. Felten. *Securing Java: getting down to business with mobile code*. Wiley Computer Pub., 2nd edition, 1999. Originally publish: Java security. 1997.
- [14] G. McGraw and G. Morrisett. Attacking malicious code: A report to the Infosec Research Council. *IEEE Software*, 17(5):33–41, Sept./Oct. 2000.
- [15] National Bureau of Standards. Data Encryption Standard. NBS FIPS PUB 46, Jan. 1977.
- [16] National Institute of Standards. The SHA-1 Secure Hash Algorithm. NIS FIPS PUB 180-1, 1995.
- [17] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL-97)*, Jan. 1997.
- [18] A. Orso, G. Vigna, and M. J. Harrold. MASSA: Mobile agents security through static/dynamic analysis. In *Proceedings of the ICSE Workshop on Software Engineering and Mobility (ICSE-2001)*, May 2001.
- [19] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, 1992.
- [20] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.
- [21] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc., second edition, 1996.
- [22] V. Shoup. On formal models for secure key exchange. Report RZ 3120 (#93166), IBM Research, 1999.
- [23] D. S. Wallach. *A new approach to mobile code security*. PhD thesis, Princeton University, Jan. 1999.