A Framework for Tamper Detection Marking of Mobile Applications[*†]

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

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

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

Abstract

Today's applications are highly mobile; we download software from the Internet, machine executable code arrives attached to electronic mail, and Java applets increase the functionality and appearance of web pages. This movement has stirred a great deal of research in the area of mobile code security. The fact remains that a newly arrived program to a local host has the potential to inflict significant damage to the local host and local resources. Perhaps the new program originated from a charlatan host masquerading as a trusted server, or has been modified by a malicious party during transit from the trusted server to the local host. In light of this risk, security models that address mobile code are in high demand. We have developed a framework named SECRYT, which enables users of a mobile application to validate the application with integrity and authentication data while simplifying the management and distribution of the authentication data.

1 Introduction

Computer software exhibits a much higher degree of mobility today than previous years, with a great amount of software being delivered to a client host via a network shortly before execution begins. The integrity of mobile code is one important aspect for its secure execution on the local host [16, 28]. We have developed a framework named SECRYT (StEgo-CRYpto Tamper detection), which enables users of a mobile application to validate the application with integrity and authentication data while simplifying the management and distribution of this data. Initial experimental results show no runtime performance degradation for the execution of the protected program.

[*]This material is based upon work supported by the National Science Foundation under Grant No. CCR-0219559.

[†]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.

A mobile code downloaded from some server to a local host could arrive from a charlatan host or be modified during transit. Once in execution on the local host, this rogue code could cause a great deal of damage to local or distributed resources and possibly compromise information integrity. The damage to the local system could be as innocuous as playing annoying sounds or as severe as denying service to some resource, deleting valuable data, or revealing secret information. The risk also exists for a malicious host to cause harm to a mobile code, altering or forging code that passes through the malicious host. Thus, a system utilizing mobile code can potentially expose itself to a great many vulnerabilities.

SECRYT embeds authentication data, which we term a Tamper Detection Mark (TDM), within the program as a way to address the issues of code integrity and authentication. SECRYT can be utilized to detect virtually any degree of tampering or alteration to an application. Validation of integrity and authentication with SECRYT is optional; a code carrying a TDM is semantically equivalent to the original version of the code and can execute without any special pre-processing should authentication of the code not be desired or should the code execute on a system that does not have an implementation of the SECRYT framework. The SECRYT framework has been implemented for Java programs [11, 12] and for SPARC binary object files.

Authentication data in SECRYT is communicated via hybrid steganographic-cryptographic (stego-crypto) techniques that marry authentication data with the program, obscure the existence of the authentication data from casual view, and do not increase bandwidth requirements. A twofold benefit is achieved by (a) encoding the authentication data within the mobile code thus eliminating the need to treat authentication data as additional information during transmission, and (b) preventing separation of authentication data from the code thus eliminating the situation where authentication data is lost or damaged which requires additional bandwidth for data retransmission.

Steganography is the process of hiding information in other information [13]. Steganography has been used throughout the ages as a means of subliminal communica-

tion. With steganography, a secret message is embedded in seemingly innocuous cover data (e.g., a picture, text, or symbols) to hide the act of communication. The embedding process usually exploits statistical randomness or noise within the data of the cover medium to hide the secret. The secret message can subsequently be delivered unnoticed because its existence is difficult to detect. Digital watermarking is a commonly known use of information hiding which is typically employed as copyright protection. Watermarking requires that the mark be robust against removal or tampering attacks to fully protect the intellectual property containing the mark. Watermarking differs from our use of information hiding which is of a fragile variety; tampering with the program will invalidate the SECRYT mark.

In the context of this paper, the terms mobile code, program, and application are used interchangeably. Whereas the term mobile code typically is defined to be code that physically relocates during the lifetime of its execution, we employ a very loose definition in that any code not compiled on the machine that is running the code is said to be mobile. Thus, almost all code can be defined as mobile code, and as the SECRYT system was designed to detect tampering with mobile code, it can be applied to almost any code.

The remainder of this paper is organized as follows. Section 2 provides context for this work by reviewing existing integrity verification techniques for mobile agents and mobile code. In Section 3, we present an overview of our system and details on its implementation. Section 4 evaluates our system through empirical study. We present research related to this work in Section 5. In Section 6, we summarize our results.

2 Background

Current popular techniques to detect/deter tampering include the use of hash digests (e.g., MD5 [19] and SHA [17] hash digest algorithms) and digital signatures (e.g., RSA [20] signature algorithm). Digital certificates [9] are also in use to attest to claims of identity on the part of a particular person or entity.

Hash algorithms such as MD5 and SHA function as a checksum to indicate the presence of errors or alterations in code received from a remote host. A good hash digest algorithm will provide for a change in 50% of the bits in the hash value should there be a 1-bit change in the code being digested. A typical hash digest check would proceed as follows: The server transmitting the code and the host receiving the code independently compute a digest of the code. These separate values will be equivalent only if the integrity of the transmitted code has not been compromised. One well known problem with relying solely on a hash digest to serve as a form of tamper detection is that the hash value can be forged, altered, or removed entirely. Digital signatures have appeared in part to answer this problem.

Digitally signed code (e.g., signed Java archives known as JAR files [15]) can address the issue of identifying compromised mobile code. The JAR file is in essence a TAR archive of a program that includes space for a digital signature of each class file of the program. When signing code, usually one computes a hash value of the code and then that hash value is encrypted and appended to the code. The encryption is usually performed with some form of a public/private key-pair. This signature can be checked on the local host side to authenticate the sender of the code and to determine if the code was altered in any way during transit. The shortfalls of signatures are: (1) the signature can become separated from the code, and (2) the signature requires extra space and bandwidth. The same risks exist with other forms of code delivery (architecture specific binary files) and authentication techniques wherein the authentication data is separate information which requires additional bandwidth during transmission.

A certificate permits one to verify the identity of the author of mobile code that has been downloaded to the local host. Under this scheme, the certificate is included with the mobile code when it is downloaded. The local host can check the contents of the certificate with the originating certificate authority (some trusted third party) to authenticate the author's identity. Certificates by themselves are lacking in several areas: (1) the certificate can be separated from the code, (2) extra time, bandwidth, and resources must be used to obtain and verify the certificate, (3) the degree of confidence one can place in a certificate is directly impacted by the integrity of the issuing authority and the degree to which that authority investigates the identity of the author, (4) a certificate can not inform the consumer about whether 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 since a certificate only verifies the authenticity of the code's originator, and (5) the certificate server provides a single point of vulnerability; if the certificate server has been compromised, the entire system is vulnerable. Issue (3) was illustrated most recently by the Microsoft Corporation certificates issued by Verisign (a certificate authority) to an unauthorized individual. These fraudulent certificates could permit the attacker to trick a user into installing malicious mobile code on the user's system [6].

3 Tamper Detection Framework

SECRYT enables application producers to embed authentication data within the program itself, thus reducing or eliminating several undesirable possibilities (e.g., executing altered or forged code on the user's machine, and retransmission of lost or damaged authentication data). Embedding the authentication data within the application (1) sim-

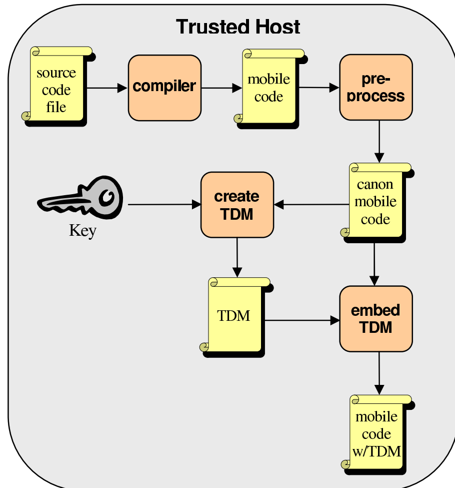

Figure 1. Embed framework on Trusted Host.

plifies management of authentication data, (2) reduces the chance of separating authentication data from the code, (3) reduces bandwidth requirements, and (4) obscures the existence of the authentication data from casual view. The savings in bandwidth is a direct result of the steganographic embedding of the authentication data within the application.

An overview of the framework for our tamper detection system, SECRYT, is shown in Figures 1 and 2. SECRYT operates in two phases, entitled embed and validate. The embed phase typically takes place on the host which compiles the source code and produces the program file. We call this host the trusted host. A basic example utilizing SECRYT follows: the trusted host compiles a program, embeds a TDM within the application, and makes the program available for download. The local host downloads the program, validates the TDM within the application via a SECRYT implementation and proceeds with execution based on the validation results.

From Figure 1, the TDM is created by transforming the application to canonical form, computing a hash value of the application and encrypting that hash value with the secret key. Since we are validating the program with a hash digest of the program, we must be sure that the local host and trusted host are synchronized by hashing a program of identical form. Embedding a TDM in the program typically results in a program of a different form, as does compiling the same program by different compilers, hence the requirement of transforming the program to what we term its canonical form before the hash value of the program can be computed.

The canonical form of a program is achieved by sorting various sections of the program based on some criteria derived from the format of executable program files for a given architecture. The specifics of canonical form for Java

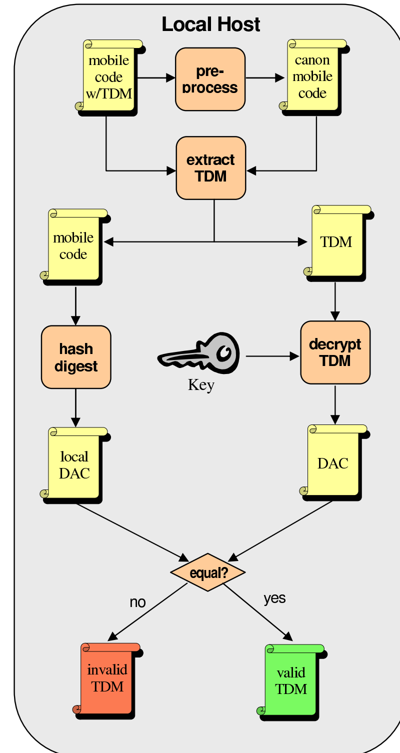

Figure 2. Validate framework on Local Host.

and binary object files (OF)s are described later in this paper. After the program has been sorted, many areas of the program file must be updated to reflect the new form of the program. The reordering and updating schemes ensure that a program in canonical form is a valid program, and can execute on a regular machine with no special pre-processing.

With the program in canonical form, the hash value of the program can be computed. The entire program file is hashed and the resulting hash value is encrypted with a secret key. This key is currently shared between the trusted host and the local host. The encrypted hash value is the TDM, thus the TDM serves as a cryptographic checksum for the program.

Embedding the TDM in an application is accomplished by permuting the order of selected sections within the application in canonical form. The permutation algorithm utilized is that given in [14]. To reorder a given section of size n within an application to encode a TDM, we select the nth permutation of all possible orderings of that section. Again, manipulating the contents of an application typically requires the entire program file to be updated to reflect the new form of the program. The embedded TDM is now part of the application. We stress that the TDM requires no additional space within the program; the TDM is encoded within the program. Program size remains constant.

Once the TDM has been embedded within the program file in the final step of Figure 1, the application is ready

for transmission to the local host. The new program file created during the embed phase is semantically equivalent to the original program file; the same computation is performed and the runtime performance of the computation should not be affected. This new program file with embedded TDM is able to execute on any machine with no special pre-processing.

Once the program has arrived at the local host from the trusted host, the local host can validate the code with a SECRYT implementation. During the validation phase, the TDM of the application is extracted, and the value of the decrypted checksum is used to validate the application. The validation phase is shown in Figure 2. The first steps of validating a TDM are similar to embedding a TDM. The application is transformed to canonical form, and a local hash digest of the program is computed. The TDM is extracted from the application by reversing the permutation algorithm. To extract the TDM, the sequence of predetermined sections of the application in canonical form are compared to their permuted sequence to determine which one of the possible permutations is encoded in their order. The extracted TDM is decrypted with the secret key and compared with the local computed hash digest. If the code has not been altered since insertion of the TDM and the proper keys have been used to create and validate the TDM, the validation result will return true. Any alteration to the code or incorrect key use will result in failure during the validation phase.

To keep the embed and validate phases as inexpensive as possible, we only permute as many elements as needed to encode the TDM. The remaining (if any) elements are scrambled randomly during the embed phase to keep the appearance of the section containing the TDM sufficiently random. The resulting reordered data has the same appearance as the normal order of the data before the embed phase.

Currently, SECRYT employs two different hash digest algorithms (depending on the available stego-bandwidth of the host program), MD5 and SHA-1. The current encryption algorithm used is 3DES. SECRYT was designed to permit both the hash and encryption functions to easily accept new algorithm implementations thereby allowing a choice of algorithms.

3.1 Instantiating the SECRYT Framework

To create a new instantiation of the SECRYT framework, several details regarding the new implementation must be thoroughly considered. In particular, the following parameters must be determined:

1. The region(s) of the file that can accommodate a TDM.
2. The unit of granularity for information hiding (i.e., permutable blocks) within the above regions.
3. The carrying capacity or stego-bandwidth available within the above regions are functions of the size of the region(s) available within the program. Does the granularity allow for adequate hiding capacity?
4. The partitioning scheme for creating the individual premutable units based on the granularity within the regions to hold the TDM.
5. The canonical form of the file
   - What properties must be met to achieve canonical form and maintain semantic equivalence with the original program?
   - How can the original program be automatically transformed to canonical form?
6. How to create the TDM.
7. How to embed the TDM within the program file.

In considering the first issue, the embedding region, the details of the target architecture must be thoroughly understood, as well as the format of the program file for the given architecture. Some areas within a program file may not be open to manipulation, while others will accommodate a great deal of modification without changing the semantics. Within the region that stores or encodes the TDM, the size or granularity of the individual units must be known. This will dictate the amount of bandwidth available to encode the mark. The order of these individual units will be permuted to encode the mark. To identify the permutable units, the region encoding the TDM may need to be explicitly partitioned. In some program files, this partitioning step may not be necessary; this step depends on the nature of the permutable unit within the program file. Once the region to encode the TDM has been identified, and the permutable units created, the nature of the program file's canonical form can be defined. The canonical form of a program file will typically involve sorting the permutable units within the region encoding the TDM and then updating the program file to reflect the new form. Once canonical form has been achieved, the TDM can be created and embedded within the program.

The TDM is embedded into this canonical form with particular attention to ensuring:

- The semantics of the program remain constant (i.e., the program performs the same computation).
- The local host is able to transform the program file to canonical form independently of the trusted host.
- The hiding capacity of the region can accommodate the size of the TDM.
- The size of the program file with TDM is no larger than the original program file.

- Extracting and validating the TDM will reveal tampering with the program file.
- The time and space to perform the transformations to canonical form, embedding and validation of the TDM are acceptable to mobile code users.

The amount of bandwidth for the TDM is determined by the size of the region encoding the mark and the number of permutable units within that region. A region with n permutable units has $n!$ unique orderings, thus can encode a value strictly less than $n!$. For example, when SHA-1 is used for the hash digest and the hash value is encrypted via 3DES (in ECB mode), we get a 192 bit TDM (160-bit hash value padded to 192 bits for three 64-bit blocks). A 192-bit TDM requires a class file with at least 47 entries (2192 = 47!). Similarly, a TDM consisting of an MD5 hash digest encrypted via DES would require 35 permutable units in the region.

3.2 Tamper Detection for Java Programs

Mobile Java programs are represented by Java class files. Each class file contains a wealth of information required to instantiate a run time object. The methods of that object are defined with Java bytecode, an interpreted, stack-based, assembly-like language.

We have instantiated SECRYT for Java in the following way. The region that encodes the TDM within the Java class file is the constant pool table. This table is an array of symbols similar in function to the symbol table of binary executable file formats (e.g., ELF [27]). The granularity of permutable units within the constant pool table is each entry within the table (constant objects) thus, no partitioning is necessary. Manipulating the order of the constant pool table objects requires the entire class file to be updated to reflect the new table ordering as there are many references to constant pool table objects throughout the entire Java class file. Once embedded, the TDM is part of the class file, represented by the ordering of the constant pool table.

The canonical form of a Java class file is created as follows:

1. The constant pool table of the class file is sorted to achieve a stable starting form.
2. All references to entries in the constant pool table are updated throughout the class file to reflect the new order of the constant pool table.

Sorting the constant pool table is a process that can be accomplished by both the trusted host which produces the class file from Java source code and by the local host which has no additional information other than the class file itself. Further, a class file with a sorted constant pool table can be achieved by a host regardless of the state the class file is in when it is received by that host. Of course, all references to constant pool table objects are updated to insure that the class file is a valid Java class file after transformation to canonical form.

Figure 3 shows the constant pool tables for the same Java class file. Figure 3(a) gives the original order of the constant pool table just after compilation by `javac`. The order of the constant pool table after encoding a TDM can be seen in figure 3(b).

Currently, SECRYT can embed a TDM in a constant pool table with 21 or more entries. As a point of comparison, the very basic "Hello World!" program has a constant pool table with 28 entries, yet an empty class (a class with no methods or data objects) has a constant pool table with 13 entries. To address these small files a shorter hash could be employed as the TDM such as the UMAC [3]. Additionally, we are exploring other areas in the class file which can accommodate encoding some of a TDM in instances of class files with very small constant pool tables.

3.3 Application to Binary Programs

Often, mobile programs are represented by binary files known as Object Files (OFs). The binary file format which we focused on for an instantiation of SECRYT is the ELF format, or Executable Linking File Format for the SPARC architecture, though the overall approach is applicable to other binary forms. The region that encodes the TDM within a SPARC object file is the text section (i.e., machine instruction section). The granularity of permutable objects within the text section is what we term a Relocatable Basic Block, which is defined below. Identifying these blocks of instructions must be accomplished by partitioning the text section. Embedding the TDM in a binary OF is accomplished by permuting the order of blocks of executable machine instructions within the text section of the OF. Manipulating the order of machine instructions requires many sections of the OF to be updated to reflect the new address of relocated instructions.

The canonical form of a binary OF is created as follows:

1. Blocks of textually consecutive instructions are identified which can be relocated.
2. The identified blocks of instructions are sorted to achieve a stable starting form.
3. All references to instruction addresses which have been relocated are updated to reflect the new address of the instruction.

The basic concept behind the system is to manipulate the order of relocatable blocks of instructions to encode a value;

```
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 3. Original and processed constant pool tables for “Hello World!” Java class file.

Table 1. Starting address for Relocatable Basic Blocks.

Block Number	Original Address	Canonical Address	Relocated Address
1	1	3	10
2	2	4	11
3	3	5	12
4	4	9	4
5	5	10	5
6	6	11	6
7	7	12	7
8	8	6	1
9	9	7	2
10	10	8	3
11	11	1	8
12	12	2	9

the TDM is stored in the text section of the mobile application. A relocatable block is similar to the basic block commonly used within the area of program analysis. A *basic block* is a linear sequence of contiguous instructions which have a single point of entry and a single point of exit for program control flow. The concept of a relocatable block builds upon a basic block in the following way: a *relocatable block* is a contiguous sequence of instructions which are terminated by a `goto` instruction (i.e., any type of unconditional jump instruction which does not store a return address). Figure 4 depicts the partitioning of a rudimentary control flow graph into Relocatable basic blocks. A control flow graph [1] is a directed graph G = (N, E) where N is a finite set of nodes (each representing a basic block) and E is a set of edges that represent the possible flow of control between basic blocks of the program. Each basic block in Figure 4 has a starting address and a sequence of instructions (i.e., stmts) associated with it. Diamond shaped blocks indicate a predicate node (e.g., an `if` statement) where control flow can branch between two paths. The set of nodes in Figure 4 are partitioned into four Relocatable basic blocks. Relocatable blocks can have multiple points of entry, and exit, but are always terminated with an unconditional jump which does not store a return address. Thus, a relocatable block can be composed of multiple basic blocks.

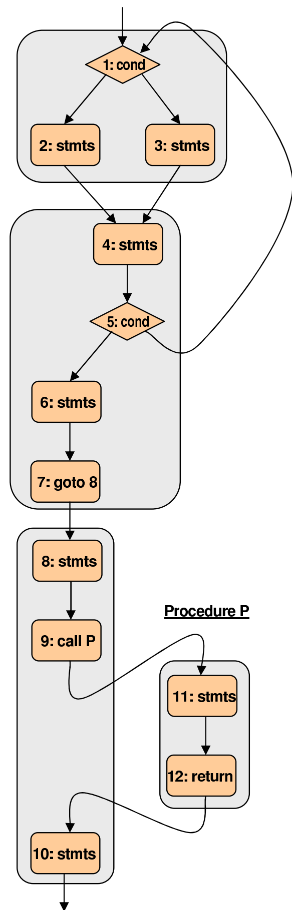

Figure 4. Graph of Relocatable basic block partitions.

Once the relocatable blocks have been identified, the process is similar to embedding a TDM in a Java class file with the exception that we are manipulating blocks of machine instructions rather than objects in the constant pool table. After constructing the relocatable blocks, the blocks are sorted according to number of instructions and instruction type to begin transforming the program to canonical form. Validation of a TDM is based on program form. Embedding a TDM within an object file changes program form, thus we must be sure that both the trusted host and the local host begin with a program of identical form. The canonical form of the object file meets this requirement.

The addresses of the Relocatable basic blocks from Figure 4 is given in Table 1. In this table, the block number is given, as is the starting address of that block within the original code (i.e, the mobile code as produced by the compiler). The third column shows the starting address of each block after the program has been transformed into canonical form. The final column shows the starting address for each block after encoding the TDM. Note that the control flow graph of the program does not change after encoding the mark; however, starting addresses of Relocatable basic blocks can change.

Moving a block of machine instructions to a new address can have a large impact on the remainder of an object file. Many sections within the file (to include the text section itself) must be updated to reflect the new address of the relocated block of code. All control structures, look-up tables, and address calculations must be appropriately updated with the new address information or the resulting object file will either fail to execute or worse yet, execute with incorrect results. The program in canonical form must be a valid program, a program which can execute correctly on a regular machine with no special pre-processing.

We are investigating other areas within the OF that can accommodate encoding pieces of a TDM. This will prove useful in instances of very small programs that contain very few machine instructions. Initial results are promising, and we have been able to locate additional hiding locations.

3.4 Security Properties

SECRYT will always be able to extract a mark from an application that contains legal machine instructions in the case of OFs and from an application that is a legal Java class file in the case of a Java application, as there is always an order to the sequence of the sections of interest. The extracted mark will either be a valid TDM or garbage (e.g., the TDM may have been scrambled by disassembly followed by recompilation yielding an invalid mark). If the extracted mark is a valid TDM, SECRYT will correctly validate the program. If the mark is garbled, SECRYT will identify the program as altered.

The security of SECRYT is reliant upon the security of the encryption and hash functions used to create the TDM. As long as the hash function employed by SECRYT has a low probability of collision, the probability of an opponent finding another program that has the same hash value as the original program is low. Compounding the difficulty of a collision attack in this unlikely event is the fact that the program that collides with the program under attack must be a legal program, or the program will fail to execute. A desirable property of any good hash function provides for a change in 50% of the bits of the hash value should there be a 1-bit change to the file. For our implementation, 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 [22]. Thus, the instantiation of SECRYT as implemented in this work has the same probability of false positives and false negatives as does SHA-1 and MD5. The encryption algorithm chosen for our proof-of-concept tool is 3DES as it was a readily available public Java library package. However, SECRYT was designed in a manner to permit both the hash and encryption functions to easily accept new algorithm implementations as they become available.

SECRYT is very sensitive to changes to TDM protected code. Even if an altered program is semantically equivalent to the original program (e.g., slight changes have been made to the application which do not affect the computation), SECRYT will signal a validation error. Any alteration, no matter how insignificant, is detectable by SECRYT.

Reverse engineering of a TDM-protected mobile code by a malicious host and subsequent recompilation will result in a code with an invalid TDM. For this kind of an attack to succeed, the malicious host must have the secret key used to create the TDM. Without the secret key, the TDM in the compromised code will not match the locally generated authentication data by the local host. During the validation phase, the local host will immediately be able to detect that the code under consideration has been altered.

SECRYT assumes that the secret key for the TDM is communicated over a secure channel prior to transmission of the mobile code. 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 the work of Shoup [23] and of Cannetti and Krawczyk [5].

4 Empirical Evaluation

TDM validation systems based on the SECRYT framework have been implemented for Java programs and for SPARC binary OFs. The Java system, called MOST (MObile Software Tamper detection), has been implemented entirely in Java and evaluated in earlier papers [11, 12]. The binary system (BMOST) has been implemented in C++, utilizing the Botan cryptographic library [4] for all hash/cryptographic algorithms. Results for the binary system are reported in this paper. All experiments were conducted on a 300MHz Sun Ultra-10 machine running Solaris8 with 192 MB of physical memory.

4.1 MOST

MOST was implemented utilizing the BCEL [8] and Cryptix [7] packages. The goal of our empirical study was to investigate the amount of time and space our technique requires to process a set of mobile Java codes. Program correctness of all TDM-protected code was also of utmost interest. We tested our implementation on the SpecJVM98 suite [25] and the Java RMI suite [18] where reported times are the medians of three execution times.

Results for embedding and validating a TDM under MOST are given in Table 2. 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 2 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 individual 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. The average time to compute a simple MD5 checksum for each class file per benchmark is 0.03 seconds. As the total size of the benchmark gets larger, the time to process the program increases as would be expected. We note that total validate time per benchmark is always strictly less than total embed time. We find this favorable as the embed process would take place on the server side, typically after compilation, whereas the validate process would take place on the client side, potentially on a lower power, mobile host.

Table 2. Embed and validate times for SpecJVM Java 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

Three Spec benchmarks (jack, javac, and mpegaudio) contain class files with constant pool tables that do not meet the current minimum size requirement of 21 entries. This restriction is being addressed in future implementations of the Java version of SECRYT.

After embedding and validation, each benchmark was checked for computational correctness. All benchmarks were found to be semantically equivalent to their original versions, with no change to run time performance.

4.2 BMOST

Table 3 shows times for embedding and validating a TDM for several benchmarks with BMOST. The benchmarks presented in Table 3 have been compiled without optimization. The size of the benchmark is given in KB as well as number of lines of source code (less comments). The number of relocatable blocks identified in the text section of the benchmark is also given. The time required to embed (including pre-process and create) a TDM as well as to validate (including pre-process and extract) a TDM is given in seconds. Transmit time across a 56 Kbps line is listed in this chart as well. Table 4 shows the corresponding times for the same benchmarks which where compiled with common program optimizations enabled. The current prototype implementation of this system targets SPARC binaries compiled via gcc. The focus on SPARC binaries from a single compiler eased implementation issues in the design of our proof of concept system.

Note that it did not take much time to process each benchmark (ranging from 0.096 seconds to 0.151 seconds for either embed or validate phases). Furthermore, no runtime penalty was observed for TDM protected codes (i.e., execution time remains constant from original to TDM-protected programs). This was a point of interest to our work as relocation of machine instructions could impact runtime performance. We hypothesize that since we are relocating only those blocks of code which are always terminated by an unconditional jump, there is minimal interference with compiler optimizations and minimal disturbance to original runtime performance.

As MD5 is a 128 bit hash algorithm and DES is a 64 bit cipher, two 64 bit blocks are required to store the encrypted hash value (TDM) in the object file. When SHA-1 is employed, the 160 bit hash value is padded to 192 bits for three 64 bit blocks. Should the object file not contain enough relocatable blocks to encode the minimum 128 bits (35 blocks) the remaining bits are hidden in other areas of the file which will not increase file size (e.g., empty pad areas, unused bits of special instructions, etc). This is a special case required for rather small programs only. We are actively searching for other areas of the object file which exhibit the required traits to accommodate steganographic manipulation.

Table 3. Embed and validate times for unoptimized SPARC code.

Bench	Size	Lines	Blocks	Embed	Validate	Xmit
wave	7.0	26	17	0.102	0.098	1.00
sort	7.5	211	17	0.099	0.099	1.07
bmm	8.8	76	21	0.102	0.101	1.26
wc	9.2	167	26	0.102	0.099	1.31
paraffins	11.0	287	57	0.108	0.109	1.57
compress	83.0	1431	97	0.151	0.151	11.86

Table 4. Embed and validate times for optimized SPARC code.

Bench	Size	Lines	Blocks	Embed	Validate	Xmit
wave	6.5	26	4	0.102	0.097	0.93
sort	6.7	211	8	0.098	0.096	0.96
wc	8.4	167	19	0.101	0.100	1.20
bmm	8.6	76	10	0.101	0.099	1.23
paraffins	9.3	287	18	0.103	0.100	1.33
compress	80.0	1431	50	0.114	0.113	11.43

5 Related Work

There are a number of techniques that address tamper detection/protection for mobile code. Tan and Moreau modified the concept of execution tracing to address mobile code (code that indeed relocates during its execution lifetime) security [26]. Under this scheme each host running a

mobile agent traces its execution of that agent. When the agent wants to move to a new host, the trace can be verified for correct execution and state via a verification server and execution certificates. Trust in specific hosts can be revoked under their distributed verification system. The new target host can accept or reject the agent based on the verification results of the verification server.

Spinellis [24] introduced the idea of utilizing reflection (the ability of a computation process to reason about itself) to provide a mobile code system with the ability to validate the integrity of deployed code on portable devices. With reflection, the code has the ability to "see" its own instructions and respond to queries about its state to some authentication server.

Hohl described a method to protect mobile agents from malicious hosts [10] for a limited time within a "black box". This technique is commonly known as program obfuscation. Obfuscation attempts to obscure program meaning, actions, and data from a potentially malicious host by introducing overly complex control structures within the agent's program instructions and by re-composing the agent's data. These obfuscating program transformations make analysis of the program more complex than analysis of the original, unobfuscated program.

Sander and Tschudin [21] have developed Computing with Encrypted Functions (CEF), which is directed mainly towards protecting code from malicious hosts. Their work demonstrates that what they call computing within a cryptographic "safe-haven" is an approach that shows some promise in the future. Currently, there is no real world implementation of this concept which addresses the general security problems faced by the consumer of mobile code. Algesheimer et al. [2] have expanded upon this approach wherein critical fragments of mobile code are executed via encrypted functions on remote hosts. A minimally trusted third party is used to perform cryptographic operations on behalf of the hosts to protect the mobile code. This approach is not practical for large applications, but may prove useful where privacy critical parts of the computation can be split out of the application for fragmented execution.

6 Conclusions

We have designed a framework, SECRYT, and implemented two systems capable of communicating program authentication data within executable code in a manner which simplifies management of the authentication data and reduces bandwidth requirements over conventionally used tamper detection techniques. These factors combine to make our system desirable in areas where bandwidth is limited (e.g., low power devices on wireless networks). TDM validation assures the user that the program was generated by a host holding the appropriate key and that the program was not altered in any way between the time the TDM was embedded and the time the user performs the validation check. Analysis indicates that our system detects with a high degree of probability, any tampering with an object file and can do so within a reasonable amount of time.

"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] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth. Cryptographic security for mobile mode. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 2–11. IEEE Computer Society, Technical Committee on Security and Privacy, May 2001.

[3] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, Lecture Notes in Computer Science, pages 216–233. International Association for Cryptologic Research, Springer-Verlag, 1999.

[4] Botan cryptographic library. http://botan.randombit.net/, 2002.

[5] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In B. Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT*, pages 451–472. Springer-Verlag, 2001.

[6] T. C. C. Center. Vulnerability note VU#869360. http://www.kb.cert.org/vuls/id/869360/, Mar. 2001.

[7] Cryptix Foundation Limited. Cryptix library. http://www.cryptix.org/.

[8] M. Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Freie Universität, 1998.

[9] A. K. Ghosh. On certifying mobile code for secure applications. In *International Symposium on Software Reliability Engineering*, page 381. IEEE, 1998.

[10] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 92–113. Springer-Verlag, 1998.

[11] M. Jochen, L. Marvel, and L. L. Pollock. Bandwidth efficient tamper detection for distributed java systems. In *Proceedings of the 16th International Symposium on High Performance Computing Systems and Applications HPCS'02*. IEEE, June 2002.

[12] M. Jochen, L. Marvel, and L. L. Pollock. MOST: A tamper detection tool for mobile java software. In *Proceedings of the 3rd Annual Information Assurance Workshop*. IEEE, June 2002.

[13] S. Katzenbeisser and F. A. P. Petitcolas. *Information hiding techniques for steganography and digital watermarking*. Artech House, 2000.

[14] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley,, 1981. 2nd ed.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[16] G. McGraw and G. Morrisett. Attacking malicious code: A report to the Infosec Research Council. *IEEE Software*, 17(5):33–41, Sept./Oct. 2000.

[17] National Institute of Standards. The SHA-1 Secure Hash Algorithm. NIS FIPS PUB 180-1, 1995.

[18] C. Nester, M. Philippsen, and B. Haumacher. Java rmi benchmark suite. http://www.ipd.uka.de/ hauma/KaRMI/benchmarks.html, 1999.

[19] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, 1992.

[20] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, Feb. 1978.

[21] T. Sander and C. Tschudin. Towards mobile cryptography. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, Technical Committee on Security and Privacy, May 1998.

[22] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc., second edition, 1996.

[23] V. Shoup. On formal models for secure key exchange. Report RZ 3120 (#93166), IBM Research, 1999.

[24] D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):51–62, 2000.

[25] Standard Performance Evaluation Corporation. SpecJVM98. http://www.specbench.org/, 1998.

[26] H. K. Tan and L. Moreau. Certificates for mobile code security. In *Proceedings of the 17th symposium on Proceedings of the 2002 ACM symposium on applied computing*, pages 76–81. ACM Press, 2002.

[27] Tools Interface Standard Committee. Tools Interface Standard (TIS) Executable and Linking Format Specification. http://developer.intel.com/vtune/tis.htm, 1993.

[28] J. Viega, T. Kohno, and B. Potter. Trust (and mistrust) in secure applications. *Communications of the ACM*, 44(2):31–36, Feb. 2001.