

MOST: A Tamper Detection Tool for Mobile Java Software

Mike Jochen, Lisa Marvel, Lori L. Pollock

Abstract—Mobile code provides a highly flexible and beneficial form of computing. However, mobile code use creates complex security considerations beyond those associated with the traditional mode of computing. This paper describes a bandwidth efficient approach to the authentication of mobile Java codes, making our tool desirable in applications where low bandwidth utilization is a requirement (e.g., wireless networks, low power devices, and distributed computation). Our tool embeds a cryptographic checksum as a tamper detection mark into Java codes so that the mark can be extracted to authenticate the source and assure the integrity of that code. We have implemented our tool in Java and evaluated its performance through empirical study. Analysis indicates that our system detects, with high probability, any degree of tampering within a reasonable amount of time, while avoiding increased bandwidth requirements.

I. INTRODUCTION

A system utilizing mobile code can potentially expose itself to a great many vulnerabilities as evidenced by the amount of research in mobile code security [1], [2], [3], [4], [5]. A mobile code downloaded 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 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. In light of these scenarios, security models that address mobile codes are in critical demand. From a security standpoint, these concerns relate to the code's well-formedness (e.g., syntax and type compatibility), behavior (e.g., file and memory access, and resource utilization), and integrity (e.g., Did the code arrive from a trusted

host? Did the code arrive intact and unchanged?).

As definitions of mobile code tend to vary, the phrase mobile code in the context of this paper is defined very loosely as any code compiled on a machine other than that which it is to execute. Under this definition, almost all current software can be classified as mobile code.

We have developed a tool called MOST (MOBILE Software Tamper detection). MOST enables a mobile software system to validate mobile Java codes with authentication data that is derived from the code itself without increasing bandwidth requirements, without the risk of separation of authentication data from code, and without relying on a third party authentication agent. MOST embeds authentication data which we term a Tamper Detection Mark (TDM), a cryptographic checksum, within the code as a way to address the issues of code integrity and authentication. This tool can be utilized within a Java environment to detect virtually any degree of tampering or alteration to a Java code. The Java platform was chosen due to its popularity as a mobile code and mobile agent language [6]. Authentication with the MOST tool is optional; a code carrying a TDM is semantically equivalent to the original version of the code without the TDM 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 is not MOST-aware.

Authentication data in MOST is communicated via steganographic techniques which decrease bandwidth requirements and obscure the existence of the authentication data from casual view. A two-fold reduction of bandwidth requirements is achieved by (1) encoding the authentication data within the mobile code thus eliminating the need to treat authentication data as additional information during transmission, and (2) preventing separation of authentication data from the code thus eliminating the situation where authentication data is lost or damaged which requires additional bandwidth to retransmit the authentication data.

Steganography is the process of hiding information in other information [7]. 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 to hide the

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.

Mike Jochen: (jochen@cis.udel.edu) University of Delaware, Newark, DE.

Lisa Marvel: (marvel@arl.army.mil) U. S. Army Research Laboratory, APG, MD.

Lori L. Pollock: (pollock@cis.udel.edu) University of Delaware, Newark, DE.

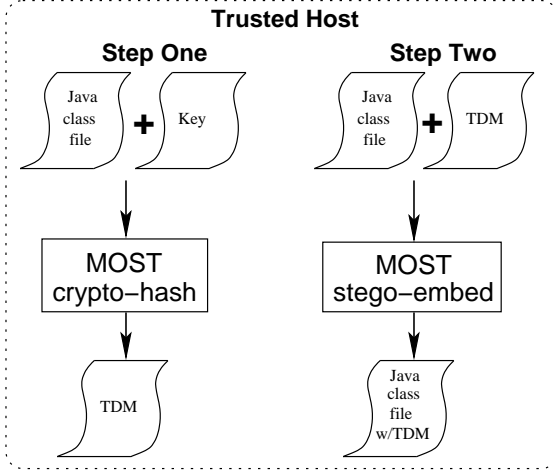


Fig. 1. Embed phase for the MOST tool.

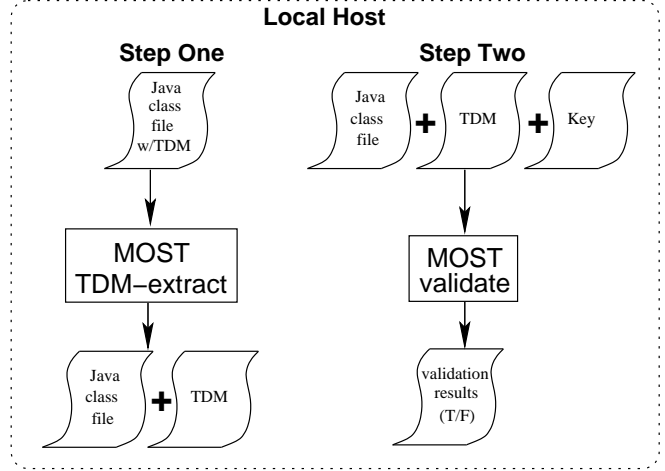


Fig. 2. Validate phase for the MOST tool.

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.

The remainder of this paper is organized as follows. In Section II, we present an overview of the MOST tool, details on its implementation, and empirical evaluation. Section III provides context for this work by reviewing existing security techniques for mobile agents and mobile code. In Section IV, we summarize and discuss future research directions.

II. THE MOST TOOL

The MOST tool provides designers and users of mobile code systems with a way to embed authentication data within the code itself, thus reducing or eliminating several undesirable aspects (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 code (1) reduces bandwidth requirements, (2) reduces the chance of separating authentication data from the code, (3) obscures the existence of the authentication data from casual view, and (4) eliminates reliance on a third party (e.g., a certificate server). The savings in bandwidth is a direct result of the steganographic encoding of the authentication data within the mobile code.

The MOST tool is designed to function either as a stand alone authentication tool or as an authentication API for inclusion in a mobile code system or framework. An overview of the framework for the MOST tool is shown in Figures 1 and 2. MOST operates in two phases, an embed phase and a validate phase. The embed phase typically takes place on the host which compiles the Java source code. We shall call this host the trusted host. A basic example utilizing MOST follows: the trusted host com-

piles a Java application, embeds a TDM within each of the application’s class files, and makes the program available for download. The local host downloads the application, validates the TDM within each class file via MOST and proceeds with execution based on the validation results.

From Figure 1, the TDM is created during Step One by computing a hash value of the Java class file and encrypting that hash value with the secret key. Each TDM serves as a cryptographic checksum for the class files that compose the mobile code.

Embedding the TDM in a Java class file is accomplished by permuting the order of the constant pool table. This table is similar in function to the symbol table of binary executable file formats (e.g., ELF [8]). The permutation algorithm utilized is that given in [9]. Manipulating the order of the constant pool table 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. The embedded TDM is now part of the class file, represented by the ordering of the constant pool table.

Once the TDM has been embedded within the code in Step Two of Figure 1, the mobile code is ready for transmission to the local host. The new class files created during the embed phase are semantically equivalent to the original Java class files; the same computation is performed and the runtime performance of the computation is not affected. These new class files with embedded TDMs are able to execute on a regular Java Virtual Machine (JVM) with no special pre-processing.

Once the mobile code has arrived on the local host from trusted host, the local host can validate the code with the MOST tool. During the validation phase, the TDM of each class file is extracted, and the values of the decrypted checksums are used to validate their respective class files. The validation phase is shown in Figure 2. To extract the

TDM, Step One requires a Java class file with embedded TDM and outputs the extracted TDM and a new class file without a TDM. Extraction of the TDM from the class file involves finding the represented permutation of the constant pool table. This process is simply the inverse of the permutation algorithm utilized during the embed phase, and yields the value of the TDM. Step Two accepts the new Java class file, the TDM, and the secret key as input. During Step Two, the value of the TDM is decrypted with the secret key and compared with a locally generated hash value. Agreement of the local and extracted hash values indicate successful code authentication. 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 a failing validation phase.

A. Security Properties

MOST will always be able to extract a mark from a legal Java code; there is always an order to the constant pool table. Any code accepted for execution by the JVM defines a legal Java code. The extracted mark from this code will either be a valid TDM or garbage (e.g., the TDM may have been scrambled by recompilation yielding an invalid mark). If the extracted mark is a valid TDM, MOST will correctly validate the mobile code. If the mark is garbled, MOST will identify the code as altered.

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

The security of MOST is reliant upon the security of the encryption and hash functions used to create the TDM. As long as the hash function employed by MOST has a low probability of collision, the probability of an opponent finding another Java code that has the same hash value as the original code is low. Compounding the difficulty of a collision attack in this unlikely event is the fact that the code that collides with the code under attack must be a legal Java code, or the JVM will halt execution and signal an exception. 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 class file. For the MOST implementation, we chose two well-known hash functions, MD5 [10] and where possible, SHA-1 [11]. 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 [12]. The encryption algorithm chosen for our proof-of-concept tool is DES [13] as it was a readily available public Java library package. However, MOST was de-

signed in a manner to permit both the hash and encryption functions to easily accept new algorithm implementations as they become available.

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 sort 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.

MOST 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 here 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 [14], [15].

B. Performance Evaluation

MOST was implemented entirely in Java, utilizing the BCEL [16] and Cryptix [17] 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. 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 [18] suite where reported times are the medians of three execution times.

We report results for seven of the ten SpecJVM98 benchmarks. Three of the benchmarks (jack, javac, and mpegaudio) contain class files with constant pool tables that do not meet the minimum size requirement of 21 entries. With a constant pool table size of 21 entries, MOST can encode 64 bits of information (DES is a 64-bit cipher). There exist $21!$ permutations of the 21 entries in the constant pool table, and $20! < 2^{64} < 21!$. We are examining techniques to address class files which contain constant pool tables with fewer than 21 entries.

Timing data for the benchmarks processed by MOST is given in Table I. Most benchmarks contained more than one class file; a single program in Java is often composed of multiple class files. Average class file sizes are given (in KB) as well as the total size of all class files for each benchmark. Table I 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 MOST performance. The last four columns indicate the average and total times

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

TABLE I
EMBED AND VALIDATE TIMES FOR SPECJVM BENCHMARKS.

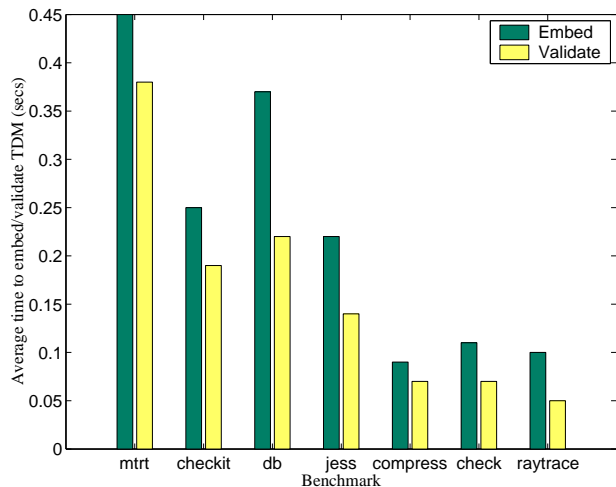


Fig. 3. Chart showing average embed and validate times (in seconds) per class file for each benchmark.

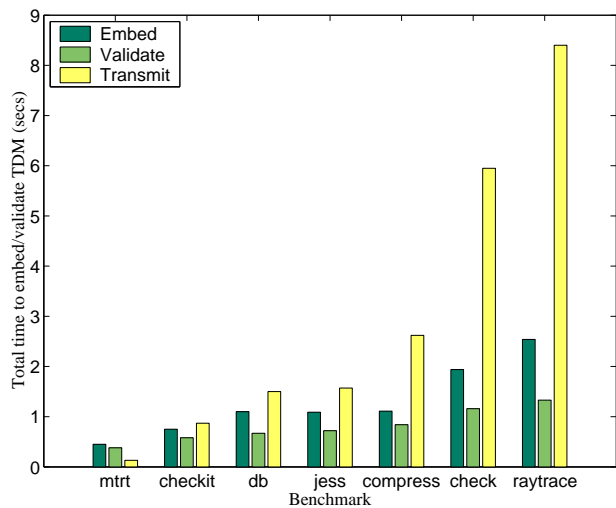


Fig. 4. Chart showing total embed and validate times (in seconds) for each benchmark.

(in seconds) for the embed and validate processes. Embed time includes time to create and embed the TDM while validate time includes time to extract and validate the TDM. The average time recorded is the average time for MOST to process each class file in the given benchmark. We note that MOST did not take much time to process each individual class file. It took an average of 0.14 seconds to create and embed the TDM and 0.09 seconds to extract and validate the TDM per class file.

Figures 3 and 4 show results of our performance testing for each benchmark. The average processing time per class file per benchmark is presented in Figure 3, while the total time to process an entire benchmark (and all associated class files) is presented in Figure 4. In both figures, the benchmarks are listed on the x-axis in order of the available stego-data bandwidth (i.e., TDM capacity). As the total size of the entire benchmark gets larger, the time to process the program increases as expected.

From Figure 3, we observed no consistent correlation between embed (validate) times and individual class file size or constant pool size per benchmark. 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 4 indicates how the total embed (validate) time increased as the size of the entire benchmark increased in our experiments. Again, the total embed time was consistently larger than the total validate time per benchmark for all benchmarks in our experiments. Lastly, the transmit time over a 53 Kbps line was significantly higher than processing time for larger benchmarks.

Both figures show beneficial properties of MOST. The time to extract and validate the TDM is always strictly less than the time to create and embed the TDM. This is a desirable property, as a potential use of MOST could involve embedding a TDM once on a trusted file server and extracting and validating the TDM possibly many times by many less powerful client machines. While network capacity is a dynamic property, and an arbitrary 53 Kbps bandwidth was chosen for comparison of our timing data,

the relation between benchmark transmission time and extract and validation time is also favorable for MOST.

After timing data was collected, processed benchmarks were executed to verify program correctness. All benchmarks with embedded TDMs executed correctly and without any change in performance. Benchmarks with TDMs were also altered to test the ability of MOST to detect various degrees of tampering. The alterations for each benchmark ranged from editing a single bit with a hexadecimal code editor to reverse engineering (decompilation, alteration, and recompilation). As was expected, any alteration resulted in a different TDM and thus, was detected by MOST.

III. RELATED WORK

There are many facets of security applicable to mobile code [4], [3]. The first consideration is to determine whether the goal is to protect the host from potentially malicious code, or to protect the code from potentially malicious hosts. A vast amount of work addresses security issues related to foreign code execution on local and remote machines and how to protect those machines. Our work addresses the issues of code integrity and authentication.

In Java, the strong typing and rigorous run-time checks performed by the JVM protect the local machine from poorly formed Java code [19]. The JVM is the component of the Java system that executes platform independent Java class files (Java programs). These checks performed by the JVM prevent against such actions as improper memory access and assignment.

Executing code in a restricted memory area is known as utilizing a reference monitor or sandbox [19]. 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, network, or printer) can be restricted to prevent a malicious mobile code from performing undesirable actions while in execution on the local machine. While the sandbox can offer some degree of protection to the user of mobile code, its presence can be rather cumbersome in defining and managing security policies for certain applets, hosts, and domains. 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) [20]. 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 certificates and signatures.

Kozen [21] and Necula [22] 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 runtime 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 user-defined, compiler generated proof. Complete automation of proof generation for more involved security policies is still an area of active research. Appel and Felten [23] 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.

A certificate permits one to validate the identity of the author of mobile code which has been downloaded to the local host [24]. 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 and the degree to which that authority investigates the identity of the author, (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.

Signed code (e.g., signed Java archives known as JAR files [19]) can address the issue of identifying compromised mobile code. Signatures do not suffer from the single point of vulnerability that certificates do. The main shortfalls of conventionally communicated signatures are: (1) the signature requires extra space and bandwidth, and (2) the signature can become separated from the code requiring additional bandwidth to retransmit the authentication data. 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. Repeated requests for retransmission of the code could follow. Removal or alteration of the more bandwidth efficient TDM from code mandates tampering with the code. The MOST system immediately alerts the destination of tampering in such instances.

Recent work has focused on shortening the length of digital signatures. Short signatures are desirable in areas where bandwidth is limited (e.g., digitally signed Internet postage, hand-keyed digital signatures, and wireless networks). Naccache and Stern [25] propose a scheme that uses a variant of Digital Signature Algorithm (DSA) and

Elliptic Curve Digital Signature Algorithm (ECDSA) to shorten the length of a signature to 30 bytes. Naccache and Stern state that further optimization can reduce the signature size to 26 bytes. Boneh, Lynn, and Shacham [26] describe a scheme that utilizes elliptic curves and produces a digital signature on the order of 20 bytes.

Sander and Tschudin [27] have completed work on Computing with Encrypted Functions (CEF), which is directed more towards protecting code from malicious hosts. This work demonstrates that what they term as computing within a cryptographic “safe-haven” is an area that shows some promise in the future. Currently, there is no real-world implementation of this idea that addresses the security problems faced by the user of mobile code. Algesheimer et al. [28] 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.

IV. CONCLUSIONS AND FUTURE WORK

We have designed and implemented a tool to enable a user to validate the integrity of mobile Java codes in a bandwidth efficient manner. Validation assures the user that the code was generated by a host holding the appropriate key and that the code was not altered in any way between the time the code was marked using MOST and the time the user performs the validation check. MOST embeds a cryptographic checksum within the code via steganographic techniques, thus eliminating the risk of separation and extra bandwidth requirements of digital signatures and certificates. These aspects make MOST an attractive form of tamper detection in wireless network systems, low power devices, and distributed computation systems. Due to its stand-alone nature, MOST does not suffer from the same vulnerabilities as utilizing a certificate server, with its single point of failure for the entire system. MOST makes code validation optional, thus validation is less obtrusive than encrypted code, which mandates additional processing of the encrypted code before execution can begin.

Analysis indicates that our tool detects, with high probability, any degree of tampering with a mobile Java code and can do so within a reasonable amount of time. This makes our tool desirable to designers and users of mobile code systems and software.

While Java is the most popular means of writing and providing mobile code, we are currently looking into expanding the set of codes that our system can handle to include mobile code and mobile agents in languages other than Java.

“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. 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.
- [2] G. McGraw and G. Morrisett, “Attacking malicious code: A report to the Infosec Research Council,” *IEEE Software*, vol. 17, pp. 33–41, Sept./Oct. 2000.
- [3] G. McGraw and E. Felten, *Securing Java: getting down to business with mobile code*. Wiley Computer Pub., 2nd ed., 1999. Originally publish: Java security. 1997.
- [4] D. S. Wallach, *A new approach to mobile code security*. PhD thesis, Princeton University, Jan. 1999.
- [5] G. McGraw and E. W. Felten, “Mobile code and security,” *IEEE Internet Computing*, vol. 2, pp. 26–29, Nov. 1998.
- [6] 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.
- [7] S. Katzenbeisser and F. A. P. Petitcolas, *Information hiding techniques for steganography and digital watermarking*. Artech House, 2000.
- [8] Tools Interface Standard Committee, “Tools Interface Standard (TIS) Executable and Linking Format Specification.” <http://developer.intel.com/vtune/tis.htm>, 1993.
- [9] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2. Addison-Wesley., 1981. 2nd ed.
- [10] R. Rivest, “The MD5 Message-Digest Algorithm.” RFC 1321, 1992.
- [11] National Institute of Standards, “The SHA-1 Secure Hash Algorithm.” NIS FIPS PUB 180-1, 1995.
- [12] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc., second ed., 1996.
- [13] National Bureau of Standards, “Data Encryption Standard.” NBS FIPS PUB 46, Jan. 1977.
- [14] V. Shoup, “On formal models for secure key exchange,” Report RZ 3120 (#93166), IBM Research, 1999.
- [15] R. Canetti and H. Krawczyk, “Analysis of key-exchange protocols and their use for building secure channels,” in *Advances in Cryptology – EUROCRYPT* (B. Pfitzmann, ed.), pp. 451–472, Springer-Verlag, 2001.
- [16] M. Dahm, “Byte code engineering with the JavaClass API,” Tech. Rep. B-17-98, Freie Universität, 1998.
- [17] Cryptix Foundation Limited, “Cryptix library.” <http://www.cryptix.org/>.
- [18] Standard Performance Evaluation Corporation, “SpecJVM98.” <http://www.specbench.org/>, 1998.
- [19] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, second ed., 1999.
- [20] M. Hauswirth, C. Kerer, and R. Kurmanowitsch, “A secure execution framework for Java,” in *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS-00)* (S. Jajodia and P. Samarati, eds.), pp. 43–52, Nov. 2000.
- [21] D. Kozen, “Efficient code certification,” Tech. Rep. 98-1661, CS Dept, Cornell University, 1998.
- [22] G. C. Necula, “Proof-carrying code,” in *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL-97)*, Jan. 1997.
- [23] A. W. Appel and E. W. Felten, “Proof-carrying authentication,” in *Proceedings of the ACM 6th Conference on Computer and Communications Security (CCS-99)* (G. Tsudik, ed.), pp. 52–62, Nov. 1999.
- [24] A. K. Ghosh, “On certifying mobile code for secure applications,” in *International Symposium on Software Reliability Engineering*, p. 381, IEEE, 1998.

- [25] D. Naccache and J. Stern, "Signing on a postcard," in *Financial Cryptography*, vol. 1962, pp. 121–135, Springer-Verlag, Feb. 2000.
- [26] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *Advances in Cryptology – ASIACRYPT 2001*, vol. 2248 of *Lecture Notes in Computer Science*, pp. 514–532, Springer-Verlag, Dec. 2001.
- [27] 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.
- [28] 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*, pp. 2–11, IEEE Computer Society, Technical Committee on Security and Privacy, May 2001.