

Towards the Safe Use of Dynamically Transformed Itinerant Software

Mike Jochen, Anteneh Addis Anteneh, and Lori L. Pollock
University of Delaware
Newark, DE 19716, USA
Email: {jochen,anteneh,pollock}@cis.udel.edu

Lisa M. Marvel
U.S. Army Research Laboratory
Aberdeen Proving Ground, MD 21001, USA
Email: marvel@arl.army.mil

ABSTRACT

Mobile code and agent-based technology is being actively investigated for use within military systems. The use of mobile code in these systems could greatly benefit future defense capabilities; however, one must first establish confidence in the secure deployment and use of mobile code before widespread acceptance of this technology occurs. This is particularly true when a mobile code is permitted to evolve or modify as it moves through a network. Dynamic program transformation or evolution can enable more efficient computation of long running programs on constrained resource hosts by optimizing the computation for the current runtime input, state, and environment. This technology can also potentially provide dynamically updated or modified program functionality. Traditional mobile code validation methods such as checksums and digital signatures will be unable to efficiently meet the security needs of this itinerant, evolving software. New validation methods must be constructed in order to allow future mobile codes to avail themselves of the advantages dynamic program modification may provide while mitigating potential security risks.

We are developing a framework and prototype system to validate mobile, dynamically-transforming code in a manner which enables the system to restrict how the code can transform as it passes through the network. This system will permit modifications to the code based on a user-defined program transformation policy. In this paper, we present the details for our framework to control dynamic program transformation. This framework is the first step towards making dynamically-transforming software a viable technology for future defense systems.

Index Terms— Mobile code, Dynamic program transfor-

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.

mation, Integrity, Program analysis, Computer security

INTRODUCTION AND MOTIVATION

Mobile code and mobile agent systems can potentially provide a wealth of new functionality, services, and benefits to future defense systems [1]–[3]. The adaptability and functionality of such systems can be increased by utilizing code or agents which evolve or modify during their execution lifetime within a network of computation nodes. Some examples of such capabilities include active networks, intelligent/autonomous agents, automated worm/virus recovery, and intelligent sensor networks [4], [5]. Before mobile code/mobile agent use is considered, one must first establish confidence in the secure deployment and use of such technology [6]–[9]. Failure to do so could result in catastrophic loss or damage to system resources, assets, or capabilities.

Traditional mobile code validation methods such as checksums and digital signatures [10]–[12] will be unable to efficiently meet the security needs of itinerant, dynamically-evolving software; the original signature or checksum becomes invalid immediately after this software evolves or is modified. New methods for validation must be developed in order to allow mobile codes to avail themselves of the advantages dynamic program modification may provide while mitigating potential security risks in an efficient manner.

This paper explores the kinds of changes that can occur in a mobile code. Changes are classified according to the nature of the change, and the potential ability to detect or to restrict that class of program change. We have used these classifications to guide the design of a framework and prototype system to validate mobile, dynamically-evolving code in a manner which enables the system to restrict how the code can transform as it passes through the network. Our framework will permit modifications to the code based on a user defined security

specification policy. Within the system, the security specification can be used by any authorized node in the network to validate proposed program transformations and ultimately, decide whether or not to transform the code. This method to validate transformations before they are applied to evolving mobile code is a proactive technology, rather than the reactive technologies currently in use. Our proactive approach will make dynamically-evolving software a viable technology for future defense systems.

EVOLVING MOBILE CODE SYSTEMS

In this paper, the term *mobile code* refers to any itinerant software that may be modified (by itself or by some other entity) as it travels through a network of computation nodes. These nodes may be interconnected via a wired or wireless network in a potentially hostile environment. The concepts of self-modifying software and mobile agents are not new [13]. While many present day examples of self-modifying software are of a malicious nature (e.g., worms and viruses), research is beginning to explore the positive benefits of this software paradigm [14].

Examples of dynamically evolving mobile code can be seen in systems that utilize just-in-time compilers (JITs) [15], [16], dynamic translators [17], and dynamic code instrumentation [18]. A JIT, typically used for interpreted languages (e.g., Java or LISP), dynamically compiles portions of a program down to native machine instructions for faster execution (because instruction interpretation is normally slower than native instruction execution). JITs are normally designed not to modify the semantic behavior of a program, but to improve performance on a given host during a particular instance of a program run. Dynamic translation attempts to increase software reuse by translating program instructions originally written for one architecture to a different target architecture at run time. Dynamic program instrumentation adds code to a program (thereby instrumenting the program) at execution time for the purpose of profiling program behavior. This enables targeted optimization and program testing based on program execution with real user input. The various benefits of any kind of dynamic modification of a program (e.g., self-modifying mobile code, JIT, and instrumented code) are increased flexibility and adaptability within the system (e.g., code optimized for current input sequences or code that learns from its environment and modifies its behavior).

Dynamic, adaptive optimization systems like Jikes RVM [19] and ADAPT [20] recompile portions of a

program during execution while applying targeted performance improving transformations to the program. The newly optimized sections of the program are swapped with the older code once they are made available by the dynamic optimizer. Research demonstrates that the performance gains of this approach make up for the overhead of performing the required analysis and the time to complete the program transformations [19]. These gains can be achieved over static compile time optimization because the dynamic optimizer has more information about the data and state of the program than does the static compile time optimizer. Voss and Eigemann showed that additional performance gains can be made by performing the transformations in parallel on a separate host while execution proceeds on the Client Node [20].

Security of Mobile Code

A vast amount of research targets security issues related to non-evolving mobile code. Current techniques include computing checksums (e.g., HMAC) [10] over the object file of the software and digitally signing the software (e.g., RSA and DSA) [11], [21]. As previously stated, these techniques apply to static software which does not evolve after computation of the checksum or signature. Any alteration to the form, state, or instructions of the program after computation of the checksum invalidates the original checksum. If every node in the network is trusted and encryption keys are managed appropriately, a new signature could be generated each time the code evolves or changes on a node. However, this approach can become cumbersome in some instances, and not possible in others (i.e., in instances where not every node in the network is able (or trusted) to have their own signing keys).

There exist new techniques to steganographically embed authentication data within a program [12], [22], [23]. These approaches take advantage of certain properties within a program to encode data which can function as a Tamper Detection Mark (TDM) without increasing program size, or altering program structure, semantics, or performance. While this novel approach is appealing as it eases management and distribution of authentication data and reduces the probability of success for certain kinds of attacks, this technique does not address instances where the program evolves as it moves through a network and the TDM is not updated. This is the same vulnerability as previously mentioned for traditional checksum/digital signature techniques.

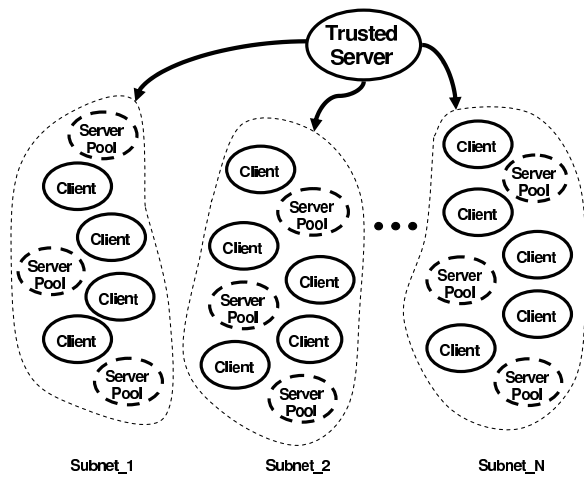


Fig. 1. Generalization of network operating environment.

TARGET NETWORK ENVIRONMENTS

Figure 1 presents a high-level view of an example of the general network environment in which our framework is designed to operate. The network will contain:

- A *Trusted Server* that distributes the original version of the program and defines the transformation control policy
- At least one *Client Node* that is considering execution of the software and possibly requesting transformations to the program
- One or more *Benign Intermediate Nodes* (i.e., other Client Nodes) that may have hosted the software in the past or requested program transformations
- Three or more *Server Pool Nodes*, which are essentially a pool of nodes, not a collection of individual nodes (we utilize the server pool concept to assist the program transformation process)
- Perhaps one or more *Malicious Nodes* that may attempt to transform the program in a nefarious manner

The network can be divided into two or more subnetworks. This network can utilize either a wired or wireless transmission medium. The configuration of the network can be either static (i.e., set once and the network does not change) or dynamic (nodes can enter/leave the network as time progresses). Client Nodes in the network trust all content from the Trusted Server. Client Nodes need only marginally trust Server Pool Nodes. This marginal degree of trust exists since the Server Pool is an ad hoc collection of nodes formed by an established promotion process. Actions taken by the Server Pool are decided by a poll of all Server Pool Nodes. All critical information for the network originates from the Trusted

Server. Content from any other source in this network (i.e., other Client Nodes) is not trusted.

The network in Figure 1 is partitioned into N distinct subnets. The classification for this grouping can be by geographic location, function of the client node, environmental conditions, or other criteria. Based upon the grouping classification, any Client Node from a given group represents all Client Nodes from the group. Within each subgroup of the network, a collection of Client Nodes functions as the Server Pool.

Server Pool technology has been devised to provide reliable services to networks of hosts [24]. The basic concept behind a Server Pool is to create a pool of several servers which provide the same service for a network. Through this pooled redundancy, service interruptions are able to be reduced. In our system, Client Nodes need only marginally trust nodes in the Server Pool.

CLASSIFYING PROGRAM CHANGES

We first characterize the nature of the changes which can occur during program transformation. In Figure 2, we present a Venn diagram classifying these changes at a high level, and depicting how these classifications relate to each other. For a given program, there can either be no change at all or some degree of change during the program's lifetime. If change is permitted during the lifetime of the program, it can take many forms. For example, the program may change only slightly (i.e., some form of *restricted change*), or it could become an entirely different program (*unrestricted change*). If program change is restricted, it could be limited by the location or region within the program, or it could be limited by the function or semantics of the program. A program that performs the same function is always *functionally equivalent*, but it could be coded in very *semantically different* ways. Within the scope of this paper, we will use the terms *change* and *transformation* interchangeably. Both of these terms will refer to any alteration to the instructions of a program through either dynamic re-optimization or recompilation.

The latter two classifications are illustrated by the following example. Consider that a programmer solves a problem with two different programs to compare which is the best solution to the problem. Program A uses algorithm a , and Program B uses algorithm b . Each program is compiled with and without compiler optimizations to compare how different optimizations affect runtime performance. Thus, we have two final versions of each program: Program A_{opt} and Program B_{opt} are programs

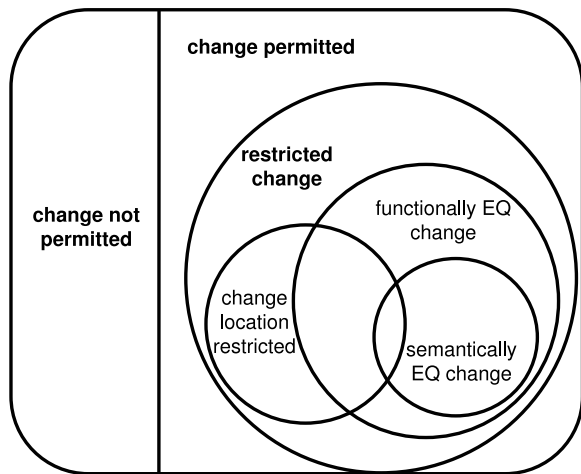


Fig. 2. High-level classification of modifications to evolving mobile codes.

compiled with optimization, and Program A_{no_opt} and Program B_{no_opt} are programs compiled without optimization. All four programs are *functionally equivalent*; they all solve the same problem. Both versions of Program A are *semantically inequivalent* to both versions of Program B ; however, Program A_{opt} is *semantically equivalent* to Program A_{no_opt} , and Program B_{opt} is *semantically equivalent* to Program B_{no_opt} .

FRAMEWORK DESIGN

The overall goal of this research is to provide a method to control or restrict how a program transforms once deployed to a network of computation nodes. There exists a delicate balance between designing an automated system which has enough flexibility to permit the kinds of program changes which provide benefit to the user, and designing a system which is powerful and robust enough to prevent programs from changing in undesirable ways, and thus allowing malicious programs to enter the system. Compounding the difficulty in designing such a system, describing program behavior and change in an automated way (i.e., by a machine) is very difficult [25], [26]. This is why many present-day methods for detecting previously categorized malicious code patterns rely on pattern matching techniques. Viewed in this light, these techniques are *reactive*, rather than *proactive* technologies; these technologies can identify a malicious behavior pattern only after that pattern has been previously identified and labeled as such. One example to illustrate this point is the use of virus definition files by anti-virus programs; these files must be continuously updated to the latest virus definitions for the virus detection software to remain effective.

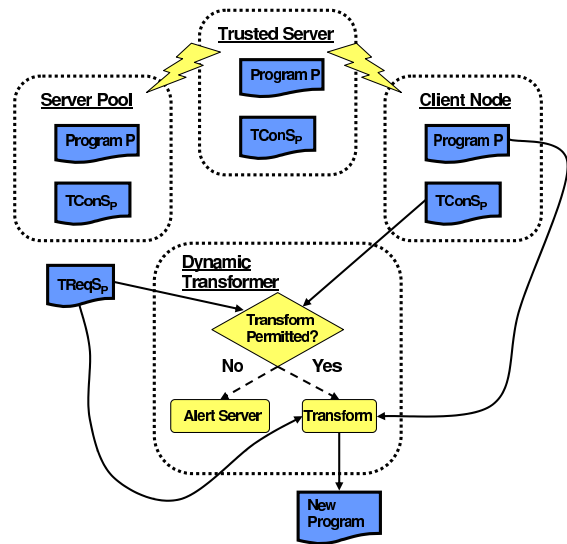


Fig. 3. A proactive program transformation control framework.

The steps towards designing a system which addresses the secure deployment and use of dynamically transforming mobile code are to (1) define the program transformation policy, (2) record and communicate that policy securely to client nodes, and (3) verify that received program transformations comply with that policy. The approach to each of these problems reveals additional issues which affect the formulation and implementation of each phase of the system.

Our approach focuses on the point before a change or transformation is to be applied to a program, and thereby functions as a pro-active technology. A generic overview of the approach where transformations are dynamically applied to a program is shown in Figure 3. The system consists of a Trusted Server, the Server Pool, and Client Nodes. Various parts of the Dynamic Transformer will reside on either the Server Pool or the Client nodes, depending on the network configuration. In this diagram, Program P and $TConS_P$ are produced by the Trusted Server while $TReqS_P$ may be produced by either the Server Pool or the Client Node.

An example scenario giving the details of our system follows. The Trusted Server publishes a program, P , to the network. P performs some critical function or computation that the Trusted Server seeks to protect from alteration. To protect this functionality or computation, the Trusted Server also publishes a Transformation Control Specification ($TConS_P$) which is associated with P . This content (P and $TConS_P$) is signed with the appropriate keys for future validation. Each host in the network receives P and $TConS_P$, validates both

objects with the appropriate keys, and proceeds with computation pending successful validation. During the execution lifetime of P , opportunities for transformation may exist that are beneficial to the system (e.g., conserve resources, etc.). These opportunities may arise based on the nature or frequency of program input, the nature of the computation, or context changes. When these opportunities occur, the request to perform this change is encoded as a Transformation Request Specification for P , ($TReqS_P$). The $TReqS_P$ is generated on the node running P and is a specification that requests specific transformations to be applied to a precise point in P . Before the transformation can be applied to P , $TReqS_P$ must be validated at the very least by the Server Pool with $TConS_P$. If the validation succeeds, the transformation is applied. If the validation fails, the Trusted Server is notified for further action and the transformation is not applied.

DETAILS OF SYSTEM CONFIGURATIONS

Since our goal for this research is to control how a program transforms on a group of networked hosts and to do so in an efficient manner, we have detailed a number of configurations which gradually shift the responsibility from one set of hosts to another. These configurations run the gamut from totally independent Client Node transformation (i.e., no Server Pool) which result in non-uniform code across the network, to complete Server Pool control over transformation. Depending on the configuration of the system/network, the following characteristics will hold: (1) the dynamic program transformer may reside either on the Client Node or on the Server Pool, (2) the $TReqS$ may be generated either by the Server Pool or by the Client Node, and (3) validation of the $TReqS$ may occur on either the Client Node, the Server Pool, or on both the Client and Server Pool. We propose to study the system configurations that provide the most efficient means of transformation control under different scenarios. The responsibilities in question are the phases of dynamic, adaptive optimization (i.e., program profiling, program analysis, and program modification).

Table I shows the entities in the network that are responsible for which actions under each configuration. Each configuration is detailed as a specific scenario. The columns for Profile and Analysis are the steps required to gather the data needed to perform program transformation. Under certain scenarios, it may be necessary to communicate or validate the $TReqS$ between the Client Nodes and the Server Pool (indicated by the columns labeled X-mit $TReqS$ and Check $TReqS$,

respectively). The entity that actually performs the program transformation is indicated by the column labeled Transform. X-mit Code denotes which entity, if any, must communicate the new (transformed) code. If the Server Pool is transforming the code, the new code must be transmitted to the Client Nodes. In this event, the Server Pool also transmits the $TReqS$ used to generate the new code ($TReqS_2$). This version of the $TReqS$ is viewed by the Client Node as a list of applied transformations. The Client Node will validate the transformation using $TReqS_2$ before accepting any new code from the Server Pool. The last column indicates whether all nodes in the network will have the same version of the program or if versions will differ across the network.

The shift in responsibilities is from the Client Nodes performing all the phases of program modification to the Server Pool taking on more of each phase. Introducing a Server Pool to perform program transformation operations requires communication that would not normally be present were each Client Node performing their own program transformation. Our studies of efficiency will include the cost of this communication.

To set up the pre-conditions for all of the scenarios, the following must occur:

- 1) The program is signed and deployed to the network by the Trusted Server.
- 2) The $TConS$ is signed and deployed to the network by the Trusted Server.
- 3) The Clients and Server Pool receive the program and the $TConS$ from the Trusted Server.
- 4) The Clients and the Server Pool authenticate the signature of the $TConS$ and the program.
- 5) The Clients and the Server Pool proceed if the Trusted Server's signature is valid.

We define three scenarios of interest to study. The first scenario, labeled Scenario 0, is a control for baseline comparison. Scenario 0 is a system without a Server Pool and without distributed dynamic transformation capabilities. In Scenario 0, all phases of dynamic transformation process occur locally on each node. The remaining two scenarios, Scenarios I and II, each have various shifts of responsibility for the phases that we will explore. In Scenario I, the Client Nodes apply the transformations. In Scenario II, the Server Pool applies the transformations. Under Scenarios I and II, the responsibilities for program analysis and generation of the $TReqS$ shifts from the Client Nodes to the Server Pool. A detailed overview of the most independent and the most controlled scenarios follow.

TABLE I

RESPONSIBILITY MATRIX. SHOWS WHICH ENTITIES ARE RESPONSIBLE FOR WHICH ACTIONS UNDER EACH CONFIGURATION.
C = CLIENT, SP = SERVER POOL.

Scenario	Profile	Analysis	X-mit TReqS	Check TReqS	Transform	X-mit Code	X-mit TReqS ₂	Check TReqS ₂	Uniform Code @ Every Node
0	C	C			C				No
I a.	C	C	C,SP	C,SP	C				Yes
I b	SP	SP	SP	C	C				Yes
II a	C	C	C	SP	SP	SP	SP	C	Yes
II b	SP	SP		SP	SP	SP	SP	C	Yes

Scenario 0

Scenario 0 is the control scenario. In this scenario, all actions for program transformation are performed by the Client Node; no Server Pool need exist. This scenario is the least communication intensive, but is the most redundant in terms of multiple clients performing the same operations (transformations). Under this scenario, there is no control over how the program transforms thus, each node on the network could have a different final version of the program. The major actions in Scenario 0 are:

- 1) Profiling is performed on the Client Node.
- 2) Analysis is performed on the Client Node.
- 3) The Client Node transforms the code.

Scenario II b

In Scenario IIb, the Server Pool performs all actions in the program transformation process. The Server Pool transmits the new code along with TReqS₂ (the applied transformations) to the Client Nodes. If the TReqS₂ successfully validates against the TConS, the new code is accepted. The major actions in Scenario IIb are:

- 1) Profiling is performed on the Server Pool.
- 2) Analysis is performed on the Server Pool.
- 3) The Server Pool generates the TReqS and validates it with the TConS.
- 4) The Server Pool transforms the code if validation is successful.
- 5) The Server Pool generates and sends TReqS₂ (TReqS) to the Client Node.
- 6) The Server Pool sends the new code to the Client Node.
- 7) The Client Node accepts the code if TReqS₂ successfully validates against the TConS.

PROTOTYPE IMPLEMENTATION

We are using the Jikes RVM [19] as the framework base of our prototype. Jikes RVM is a research virtual machine that performs dynamic, adaptive optimization

on Java programs. Our system partitions Jikes RVM into distinct phases which can be evoked and controlled via the TReqS and TConS specifications. The partitioned phases of the Jikes RVM serve to function as the various phases which would reside on the Client Nodes and the Server Pool. Initial simulations indicate that we are able to prevent or control what transformations are applied at a specific point within a program in Jikes RVM.

Program transformation is permitted/prohibited during the optimization phase of the Jikes RVM by consulting the TConS before the transformation is applied to the program. During each phase of the optimizer, a specific transformation is applied at specific points within the program. As the optimizer prepares to perform a given instance of a transformation, it performs a lookup in the TConS to see if that transformation is permitted at that point in the program.

EVALUATION

The metrics we are using to assess efficiency will examine cost per subnet under each scenario. Here, an inverse relationship is favorable (i.e., lower cost means greater efficiency). Cost will be measured in terms of the following criteria: (1) power, (2) space (memory), and (3) time. Cost will be studied based on a ratio of the number of Server Pool nodes to the number of Client Nodes in the subnet.

We will measure the cost to:

- Profile the program at runtime
- Analyze the runtime behavior of the program via the profile data and the program's instructions
- Optimize the program
- Transmit the specifications (from the Client Node to the Server Pool or from the Server Pool to the Client Node)
- Transmit the transformed code (from the Server Pool to the Client Node)
- Generate the TReqS (on either the Client Node or the Server Pool) or the TReqS₂ (on the Server Pool)

- Validate the TReqS (on either the Client Node or the Server Pool) or the TReqS₂ (on the Client Node) with the TConS

CONCLUSIONS

In this paper, we have introduced the issues, the overall design configurations, and the evaluation plan for a system to validate evolving itinerant software. Future defense systems could benefit greatly from this kind of code once the risks of its use have been mitigated. A program transformation control specification defines the system's rules on how the program may/may not transform after deployment. Experimental evaluation will explore the efficiency and expressiveness of the policy for change as communicated through the TConS by the trusted server. This system will permit users of mobile, evolving code to take advantage of the benefits this paradigm has to offer while providing the opportunity to control how the program evolves in the network.

"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] C. Rouff and W. Truszkowski, "A process for introducing agent technology into space missions," in *Proceedings of the IEEE Aerospace Conference*. IEEE, 2001.
- [2] A. Quan, R. Crawford, H. Shao, K. Knudtzon, A. Schuler, D. Scott, S. Hayati, R. Higginbotham, jr., and R. Abbott, "Automated threat response using intelligent agents (ATIRA)," in *Proceedings of the IEEE Aerospace Conference*, 2001.
- [3] D. Kotz, G. Jiang, R. Gray, G. Cybenko, and R. A. Peterson, "Performance analysis of mobile agents for filtering data streams on wireless networks," in *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. ACM Press, 2000, pp. 85–94.
- [4] S. Eichert, O. N. Ertugay, D. Nessett, and S. Vobbilisetty, "Commercially viable active networking," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 1, pp. 8–22, 2002.
- [5] S. Sidiroglou and A. D. Keromytis, "Countering network worms through automatic patch generation," *IEEE Transactions on Security and Privacy*, 2005.
- [6] E. Bierman and E. Cloete, "Classification of malicious host threats in mobile agent computing," in *Proceedings of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology*, 2002.
- [7] P. T. Devanbu and S. Stubblebine, "Software engineering for security: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*. ACM Press, 2000.
- [8] D. M. Chess, "Security issues in mobile code systems," in *Mobile Agents and Security*, ser. Lecture Notes in Computer Science, G. Vigna, Ed. Springer-Verlag, 1998, vol. 1419.
- [9] G. McGraw and G. Morrisett, "Attacking malicious code: A report to the Infosec Research Council," *IEEE Software*, vol. 17, no. 5, pp. 33–41, Sept./Oct. 2000.
- [10] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," RFC 2104, 1997.
- [11] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public key cryptosystems," *Communications of the ACM*, Feb. 1978.
- [12] M. Jochen, L. Marvel, and L. L. Pollock, "A framework for tamper detection marking of mobile applications," in *Proceedings of the Fourteenth International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Nov. 2003.
- [13] N. M. Karnik and A. R. Tripathi, "Security in the Ajanta mobile agent system," *Software-Practice and Experience*, vol. 31, no. 4, pp. 301–329, Apr. 2001.
- [14] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto, "Exploiting self-modification mechanism for program protection," in *Proceedings of the 27th Annual International Computer Software and Applications Conference*. IEEE, 2003.
- [15] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth, "Practicing JUDO: Java under dynamic optimizations," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00)*, ser. ACM Sigplan Notices, vol. 35.5, June 2000, pp. 13–26.
- [16] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. ACM Press, 2000, pp. 1–12.
- [17] K. Ebcioglu and E. R. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ACM SIGARCH and IEEE Computer Society TCCA, June 2–4, 1997, pp. 26–37.
- [18] M. M. Tikir and J. K. Hollingsworth, "Efficient instrumentation for code coverage testing," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM Press, 2002, pp. 86–96.
- [19] M. Arnold, M. Hind, and B. G. Ryder, "Online feedback-directed optimization of Java," in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2002.
- [20] M. J. Voss and R. Eigemann, "High-level adaptive program optimization with ADAPT," in *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*. ACM Press, 2001, pp. 93–102.
- [21] National Institute of Standards and Technology, "Digital signature standard," NIST FIPS PUB 186, 1994.
- [22] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding information in program binaries," in *Proceedings of the 6th International Conference on Information and Communications Security, ICISA*. Springer-Verlag, Oct. 2004.
- [23] M. Jochen, L. Marvel, and L. L. Pollock, "Tamper detection marking for object files," in *Proceedings – IEEE Military Communications Conference MILCOM*. IEEE, Oct. 2003.
- [24] M. A. Fecko, U. C. Kozat, S. Samtani, M. Ümit Uyar, and I. Höklek, "Reliable and dynamic access to service in battlefield ad hoc networks," in *Proceedings – IEEE Military Communications Conference MILCOM*. IEEE, Nov. 2004.
- [25] M. G. Pleszkoch and R. C. Linger, "Improving network system security with function extraction technology for automated calculation of program behavior," in *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*. IEEE, 2004.
- [26] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 11th USENIX Security Symposium*, Aug. 2003, pp. 169–186.