Enabling Control over Adaptive Program Transformation for Dynamically Evolving Mobile Software Validation* †

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

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

ABSTRACT

Many researchers are investigating the use of adaptive program transformation as a way to efficiently improve program performance. Performance improving transformations are performed at runtime to adapt to the possibly changing runtime characteristics of the program. Leveraging this kind of program transformation on multiple hosts can achieve these same performance gains while reducing the overhead to apply the transformations on the local machine running the program. The reduction in overhead is obtained by distributing the responsibilities for the transformation process to multiple hosts throughout the network. The use of this technology could greatly benefit applications running on networked computation nodes; however, one must first establish confidence in the secure generation and distribution of the transformed versions of the original program before acceptance and execution can occur for many network environments.

Since programs are being transformed dynamically, traditional program validation methods such as checksums and digital signatures will be unable to efficiently meet the security needs of this possibly itinerant, transforming software. New validation methods must be developed in order to allow future software to avail itself of the advantages that dynamic program modification may provide while mitigating potential security risks. In this paper, we present our framework to validate dynamically-transforming software in a manner that enables the system to restrict how the software can transform as it executes on a network of hosts. Our prototype system utilizes specification languages to communicate program transformations and controls for those transformations on hosts in the system. This first step towards validating evolving mobile code before transformation occurs, will make dynamically-transforming software a safe and viable future technology.

*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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Keywords

Mobile code, Dynamic and Adaptive Program Transformation, Integrity, Program analysis, Computer security

1. INTRODUCTION AND MOTIVATION

It has been shown that adaptive optimization of programs during execution can provide an appreciable performance boost [2, 3]. With this kind of execution scheme, programs are transformed during execution to optimize their performance based on their current runtime input, state, and environment. This approach can generate a much more specialized or highly-tuned version of a program than the traditional static compilation/optimization paradigm is able to produce. For clusters of networked hosts, performing the steps of adaptive optimization in task-parallel can provide even greater performance gains [29]. These gains are achieved by allowing the computation to continue on one host while transforming it in parallel on another.

Applications often execute within untrusted network environments. Dynamic program transformation within this type of environment can introduce the same security concerns that exist when allowing mobile code or programs to run on local hosts. The problem of mobile code security is one that has been actively studied, and one which remains an active area of interest [5, 8, 12, 22]. Before adopting dynamic program transformation technology in an untrusted environment, one must first establish a method to validate the transformations that are to be applied to the program. Failure to do so could result in catastrophic loss or damage to system resources. In some network environments, it may not be possible to establish complete trust in all hosts. For this reason, there must exist a way for a host to accept or reject dynamic optimizations (or transformations) that are to be applied to a program. The decision to accept or reject program transformations must be based on a well-defined policy for the program.

Traditional mobile code validation methods such as checksums and digital signatures [17, 21, 27] will be unable to efficiently meet the security needs of dynamically evolving software; the original signature or checksum becomes invalid immediately after the software evolves or is modified. New methods for validation must be developed in order to allow future mobile codes to avail themselves of the advantages that dynamic program transformation may provide while mitigating potential security risks. In networks with

constrained resources, this must all be done in the most efficient manner possible.

In this paper, we present our framework to validate dynamically-transforming software in a manner that enables the system to restrict how the software can transform as it executes on the network of hosts. One example application environment for such a system would be a network of low power remote sensor devices connected via a wireless network. Such networks are being deployed for interests as varied as military, commercial and research applications. Another example environment would be intelligent devices connected via a Bluetooth network [6].

Our framework utilizes specification languages to communicate program transformations and controls for those transformations within the network. This framework will allow the client nodes within the network to refer to a transformation control policy before accepting any proposed changes to a program.

The particular contributions of this paper are:

- Generalization of adaptive, distributed program transformation from a closed, trusted environment to a more open, potentially malicious environment
- Formal identification and presentation of the issues posed by adopting this paradigm of distributed, adaptive program transformation
- Presentation of our framework to address the identified issues
- Definition and specification of a language to control and request dynamic program transformations within our framework

For the purposes of this paper, the term *mobile code* and *mobile agent* both refer 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 [20]. 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 [19].

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, 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 [9, 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.

The remainder of the paper is organized as follows. In Section 2, we provide relevant background material for this problem. We present an overview of our framework Section 3. We then provide details on specification languages, and system implementation in Sections 4, 5, and 6. Lastly, we conclude with a discussion of related work in Section 7, and a summary of our contributions in Section 8.

2. PROBLEM BACKGROUND

Evolving Software and Security

An example of dynamically evolving mobile code can be seen in systems that utilize just-in-time compilers (JITs) [4, 10], dynamic translators [13], and dynamic code instrumentation [28]. 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).

Adaptive optimization systems like Jikes RVM [3] and ADAPT [29] 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 [3]. 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 [29].

A vast amount of research targets security issues related to non-evolving mobile code. Current techniques include computing checksums (e.g., HMAC) [21] over the object file of the software and digitally signing the software (e.g., RSA and DSA) [24, 27]. 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 [14, 17, 18]. 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

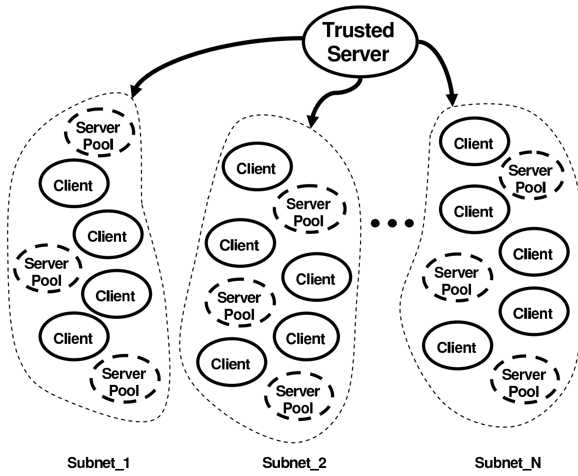

Figure 1: Generalization of network operating environment.

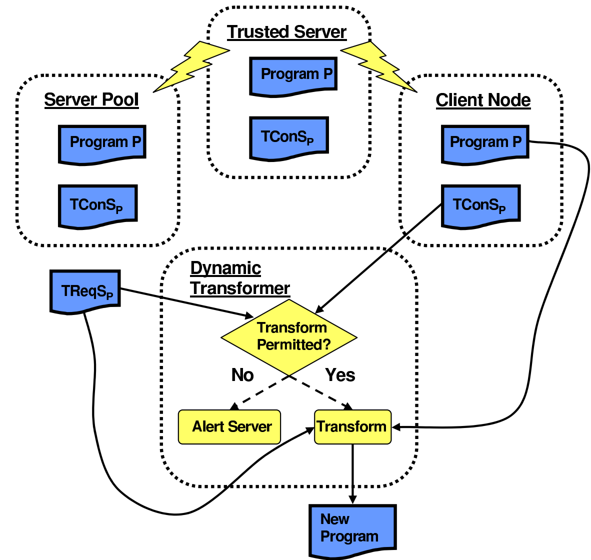

Figure 2: A proactive program transformation control framework.

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.

Network Environment

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 designed to assist the program transformation process – the Server Pool is treated as a single entity, not a collection of nodes
- 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. 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 subgroups. 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 [15]. 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.

3. CONTROL FRAMEWORK

In our approach, 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.

Depending on the goals and the security requirements of the system, two different philosophies in approach can be taken when defining the program transformation policy for change. A *permissive* approach would, by default, permit all modifications to a program, unless explicitly disallowed. This approach would foster the most creative, flexible environment for program change, but could create the highest security risk for the system. Conversely, a *restrictive* approach would disallow all program transformations, unless explicitly allowed. A restrictive approach could generate policies that are easier to verify than a permissive approach and create less risk for the system while potentially stifling program change.

Our approach focuses on the point before a change or transformation is to be applied to a program, and thereby functions as a proactive technology. A generic overview of the approach where transformations are dynamically applied to a program is shown in Figure 2. The system consists of a Trusted Server, the Server Pool, and Client Nodes. Various components of the Dynamic Transformer reside on either the Server Pool or the Client nodes, depending on the network configuration.

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. 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 the change is encoded as a Transformation Request Specification for P, (TReqS_P). The TReqS_P is generated on the node running P (e.g., a Client Node or a node within the Server Pool) 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 and the transformation is not applied.

4. SPECIFICATION LANGUAGE DETAILS

Formal specification is the main vehicle to communicate the policies and requests for program transformation in our framework. We have developed a language to handle communication of the control and requests of program transformations (TConS and TReqS, respectively).

```
<specification> :=
  TCONS <class_name> {
    <method_rule>
  }

<method_rule> :=
  /* epsilon (empty production) */
| METHOD <method_name> {
    <transformation>
  }

<transformation> :=
  /* epsilon (empty production) */
| <transformation_name> {
    address(<addrange>,
    <transformation_rules>);
  }

<transformation_rules> :=
  /* epsilon (empty production) */
| /* <transformation
     specific rule
     productions> */

<class_name> :=
  /* identifier token */

<method_name> :=
  /* identifier token */

<addrange> :=
  <adress> : <address>

<address> :=
  /* address token */
```

Figure 3: Grammar for TConS language

In this section, we present the grammar for the TConS language and some examples of its use. Figure 3 shows a simplified grammar for the TConS language. In this figure, productions for the language take the form of <non_terminal> := *production*, where *production* can be zero or more terminals or nonterminals representing other productions. The symbol "|" is used to indicate the disjunction "or" for a compound production. From Figure 3, we see that a TConS specification takes the general form:

```
TCONS
"class_name" {
  "method_name" {
    "transformation" { "specific_rules" }
  }
}
```

The "`TCONS`" keyword tells the system that this is a specification to control program transformations. The remainder of the specification is composed of blocks of text within curly braces ("{" and "}") much like any C, C++, or Java program. Transformations can be specified on a class, method, or address level. The initial transformations that this system is being designed to handle are:

- Constant propagation
- Copy propagation
- Method inlining
- Dead code elimination
- Scalar replacement of aliases
- Common subexpression elimination
- Right-hand side of assignment statement transformation

With one exception, the above transformations are those commonly employed in optimizing compilers and defined in [23]. The "right-hand side" transformation is a generalization of any program transformation which affects the right-hand side of any assignment statement to a variable of interest. Additional transformations will be added as the system matures.

Figures 4 and 5 give an example Java program and its associated TConS specification. The statements of each method from the example program in Figure 4 have been labeled with contrived addresses, shown along the left side. The addresses given in this example represent the address of the corresponding sequence of bytecode instructions.

The TConS specification shown in Figure 5 defines policies for methods `main` and `subtract` of class `foo`. The specification allows copy propagation of the value in variable `w` (unless being assigned to `y`) and of the variable `x` (only if being assigned to `z`) between address 0 – 36 in method `main`. Within method `subtract`, copy propagation is permitted for variable `a` and variable `b` (only if assigning `b` to `c` or `d`) between address 0 – 20.

Currently, the TConS specification is generated by hand after the program is compiled to Java class files. One future goal for this work is to automate this process through interaction within an integrated programming environment where a programmer can select regions of program source code to apply high-level abstractions of transformation restrictions. Detailed restrictions for individual transformations will always be permitted in this system; however, we do not want to require that sort of low-level involvement from the programmer.

An example application and generation of the TConS specification could proceed as follows. A program is deployed to the network which contains one particular value, computation, or operation that the Trusted Server wants to protect. To generate a policy

```
     public class foo {

       public static void
 0     main(String args[]) {
 4       int w, x, y, z;
 8       w = 5;
12       x = 10;
16       y = w;
20       z = x;
24       System.out.println(
            "The answer is: " +
            subtract(w, z) );
       }

       public static int
 0     subtract(int a, int b) {
 4       int c, d;
 8       c = a;
12       d = b;
16       return c - d;
       }
     }
```

Figure 4: Example Java program

```
TCONS Foo {
  METHOD main {
    COPY_PROP {
      /* At address 0 - 24, may propagate w
         unless to y, may propagate x to z */
      address(0:24, w(), !w(y), x(z) );
    }
    CONSTANT_PROP {
      /* At address 8 - 24, no propagation
         to y, may propagate to x if
         constant is between 1 - 199 */
      address(8,24, !y(), x(1:999) );
    }
  }
  METHOD subtract {
    COPY_PROP {
      /* At address 0 - 16, may propagate
         a anywhere and b to c and d */
      address(0:16, a(), b(c, d) );
    }
  }
}
```

Figure 5: Example TConS specification

that prevents any alteration to that value or operation, a backward slice[1] of the program is generated, and all transformations to the values/operations contained within that slice are prohibited.

We are designing the grammar in a generic enough manner to be able to control many kinds of program transformation and to request individual or specific program transformations. For now, the system is being designed to handle only those program transformations as implemented by the Jikes RVM, the basis for our prototype implementation. As such, the TReqS is merely the optimization level passed to the optimizer (i.e., level 0, 1, or 2 within Jikes RVM).

5. PROTOTYPE IMPLEMENTATION

We are using the Jikes RVM [3] as the framework base in our prototype. Jikes RVM is a research virtual machine that performs 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 transformations that 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.

[1]Program slicing is a program analysis technique that, for a given point and value in a program, examines the source code of that program and produces a listing of all the statements that either affect the computation of the value at that point (i.e., a backward slice) or that are affected by the value from that point on (i.e., a forward slice) [16].

6. A NOTE ON TRUST

Various configurations on how nodes interact within the network will have different ramifications on trust. Assuming Client Nodes will not transform their own software in a nefarious manner, each node can trust the transformed versions of the program, should individual Client Nodes have total control over the transformations to the program. However, this creates an environment where each node could be executing different versions of a program, possibly yielding different results. This is especially true when the notion of program transformation is generalized to permit transformations that do not preserve program semantics. In some situations, this may be an undesirable situation.

Depending on the role of the Server Pool, Client Nodes may have to place more trust in the credentials of the Server Pool. This is especially true as we shift the responsibility for TReqS generation away from Client Nodes and towards the Server Pool.

Should the Server Pool provide TReqS for the Client Nodes, the Client Nodes have the ability to validate the transformations before they are applied. If the transformations fail validation, then the Client Node will not apply those transformations. Should the Server Pool apply the transformations and provide new (transformed) software to the Client Nodes, the Client Nodes have two methods to validate the transformed software. The first method available to the Client Node is to validate the list of transformations the Server Pool states was performed with the TConS. In this instance, the Client Nodes must accept the fact that the individual nodes in the Server Pool unanimously agreed on the new form of the program. There will be a group signature from the Server Pool to assure this. When viewed in this way, a balance of trust must be taken into consideration in the shifting of responsibilities of program transformation from the Client Nodes to the Server Pool. While this introduces the complexity of a group signature, restricting this signature to a subset of network nodes (i.e., the Server Pool) eases the tasks of key management and group signature generation.

If the Server Pool colludes against the Client Node and provides transformed code with transformations not included in the transformation list, this first method of validation will fail to indicate this. To prevent this, Client Nodes can, at will, reproduce the trans-

formations given in the transformation list and compare the result with the transformed code as received from the Server Pool. If this comparison shows the two codes are the same, the Client Node knows the transformed software is valid. If these two codes are different, the transformed software is invalid and the Client Node has evidence of Server Pool collusion. This is somewhat similar to detecting misbehavior of nodes in ad hoc networks [7]. Here, when a Client Node detects a misbehaving Server Pool, this evidence can be given to the Trusted Server to revoke the credentials of the Server Pool nodes. In the absence of evidence or suspicion of misbehavior of the Server Pool, Client Nodes will reproduce the Server Pool's transformations at random intervals for comparison. When suspicion of this misbehavior exists, this transformation reproduction will always be conducted.

When the Server Pool is combining multiple TReqSs from Client Nodes to create one, amalgamated TReqS, well-defined rules must exist to address inconsistencies or conflicts between requests from different Client Nodes. When gathering TReqSs from different hosts, the possibility exists for two (or more) hosts to request a conflicting version of a program transformation. Research has explored program transformations, their effect, and their necessary preconditions [30]. We will leverage this work to assign weights to transformations based on heuristics to decide the highest priority transformations that should be applied, and how to resolve conflicting TReqSs.

7. RELATED WORK

Necula's work on Proof-Carrying Code (PCC) [25], and other's extensions to PCC [1, 11] provide a method for the safe execution of untrusted code. To utilize the PCC concept, the client provides a set of safety rules that are required to be met before execution is permitted on the client host. The code producer uses these rules to generate a safety proof that the client can use to verify that the code does not violate any rules. These proofs are bundled with the program and delivered to the client. Work continues on PCC to make a system that is more secure, more expressive (in terms of safety policies), and fully automated (in terms of proof generation and verification). PCC will not guarantee the final computation of a program meets some criteria, but PCC will verify that certain behaviors of a program during execution will adhere to a specified policy. Our work attempts to address changes in a trusted program whereas PCC addresses program behavior. Under PCC, a program altered after proof generation results in one of three possible outcomes: (1) the proof will no longer be valid and the program will be rejected, (2) the proof will be valid but will not be a proof for the program and the program will be rejected, or (3) the proof will be a valid safety proof for the program and the program will be accepted. In the third example, the program will adhere to the safety rules, but program behavior can change. Our work attempts to assist in deciding if a new version of a program should be permitted access to a computation node.

Pleszkoch and Linger describe a technique which they term function extraction to automatically extract program behavior from a program's source code in order to detect malicious program behavior [26]. This technique employs a function-theoretic model of software which treats programs as mathematical functions which map a given domain to a range. Function extraction generates behavior signatures for every control structure within the program. These behavior signatures abstract away computation detail and represent the essence of the computation. The behavior for a program is the composition of the behavior of all procedures, which are in turn compositions of all control structures within their respective procedures. Thus, program behavior is extracted in a bottom-up fashion. Function extraction can assist with program understanding and engineering of large-scale software systems. Future research on function extraction will target identifying behavior signatures of malicious code, handling loops within a program, and handling data pointers and aliases appropriately. This differs from our work in that their work is searching exclusively for known bad behaviors in programs.

Christodorescu and Jha present a method of static analysis for executables in order to detect malicious patterns of behavior in the program [9]. The tool that the authors present transforms executable forms of a program into an intermediate representation (via disassembly). The intermediate representation is statically analyzed to construct an annotated Control Flow Graph (CFG). This graph is compared with a Malicious-Code Automaton (MCA) in such a way as to mitigate the effects of obfuscation on the malicious code. The comparison is performed by treating the CFG and the MCA as languages and determining if the intersection of these languages is non-empty. A non-empty intersection indicates that the MCA is contained within the CFG and thus, the program. MCAs are built from libraries of known viruses. This technique appears promising, being able to detect versions of known viruses which have undergone transformations that render current virus detection software ineffective. This differs from the problem of controlling dynamic program transformation, which our work addresses.

Voss and Eigemann present a method of automated, de-coupled, adaptive program transformation in [29]. In this work, the authors de-couple the phases of dynamic adaptive transformation to parallelize the process. In this way, the authors can improve upon the performance gains of dynamic adaptive transformation by allowing the local host to continue computation while another processor or host is performing program optimization. When new (optimized) versions of the program are available, they are replaced on the local host. We are extending this work to allow hosts to control how a program is transformed based on some pre-defined security policy and to allow the analysis from multiple, homogeneous hosts to be used as input on what transformations should be performed. Our work seeks to allow such systems to be executed in open network environments.

8. CONCLUSIONS

In this paper, we have introduced the issues and a proposed framework for validating dynamically transforming software. The framework leverages the benefits of dynamic, adaptive optimization with the ability to control how a program is transformed. Future systems could benefit greatly from this kind of software once the risks of its use have been mitigated. Transformation specifications are used to represent the requested and permitted transformations within a program. Experimental evaluation will explore the efficiency and expressiveness of the policy for change as communicated through TConS and TReqS.

The specific contributions of this paper are: (1) the generalization of adaptive, distributive program transformation from a closed, trusted environment to a more open, potentially malicious environment, (2) formal identification and presentation of the issues involved in adopting this paradigm of distributed, adaptive, dynamic program transformation, (3) presentation of our framework to address the associated issues, and (4) definition of a language specification to control and request dynamic program transformations within our framework. Systems built with our framework will permit users of mobile, transforming 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. Our work is an initial step in making this kind of software safer to use, thus a

viable choice for software designers of distributed or network computation systems.

"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."

9. REFERENCES

[1] A. W. Appel. Foundational Proof-Carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS-01)*, pages 247–258, Los Alamitos, CA, June 16–19 2001. IEEE Computer Society.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 47–65. ACM Press, 2000.

[3] 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*, pages 111–129. ACM Press, 2002.

[4] 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*, pages 1–12. ACM Press, 2000.

[5] E. Bierman and E. Cloete. Classification of malicious host threats in mobile agent computing. In *Proceedings of the 2002 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology*, pages 141–148. South African Institute for Computer Scientists and Information Technologists, 2002.

[6] Bluetooth SIG. Specification of the Bluetooth System. http://www.bluetooth.org, 2003.

[7] A. A. Cardenas, S. Radosavac, and J. S. Baras. Detection and prevention of mac layer misbehavior for ad hoc networks. Technical Report SEIL TR 2004-4, University of Maryland College Park, 2004.

[8] D. M. Chess. Security issues in mobile code systems. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[9] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 11th USENIX Security Symposium*, pages 169–186, Aug. 2003.

[10] 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)*, volume 35.5 of *ACM Sigplan Notices*, pages 13–26, June 2000.

[11] C. Colby, K. Crary, R. Harper, P. Lee, and F. Pfenning. Automated techniques for provable safe mobile code. *Theoretical Computer Science*, 290:1175–1199, 2003.

[12] 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.

[13] K. Ebcioğlu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 26–37. ACM SIGARCH and IEEE Computer Society TCCA, June 2–4, 1997.

[14] 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.

[15] 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.

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[17] 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.

[18] M. Jochen, L. Marvel, and L. L. Pollock. Tamper detection marking for object files. In *Proceedings – IEEE Military Communications Conference MILCOM*. IEEE, Oct. 2003.

[19] 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.

[20] N. M. Karnik and A. R. Tripathi. Security in the Ajanta mobile agent system. *Software–Practice and Experience*, 31(4):301–329, Apr. 2001.

[21] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, 1997.

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

[23] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 2000.

[24] National Institute of Standards and Technology. Digital signature standard. NIST FIPS PUB 186, 1994.

[25] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL-97)*, Jan. 1997.

[26] 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.

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

[28] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 86–96. ACM Press, 2002.

[29] 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*, pages 93–102. ACM Press, 2001.

[30] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, Nov. 1997.