EXPERIENCES IN DESIGNING A SYSTEM TO CONTROL DYNAMIC PROGRAM TRANSFORMATIONS

Mike Jochen
East Stroudsburg University of Pennsylvania
Email: mjochen@esu.edu

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

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

ABSTRACT
Adaptive program optimization during execution can provide appreciable performance boost with respect to the possibly changing runtime characteristics of the program (e.g., runtime input and state). This approach can generate a much more highly-tuned version of a program than that produced by the traditional static compilation/optimization paradigm. For clusters of networked hosts, performing the steps of adaptive optimization in task-parallel can provide even greater performance gains.

Applications often execute in untrusted network environments. Distributed, dynamic program transformation within this type of environment can introduce the same security concerns that exist when allowing mobile code to run on local hosts. Before adopting dynamic program transformation technology in an untrusted environment, one must first 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. For this reason, there must exist a way for a host to accept/reject dynamic transformations that are to be applied to a program. The decision to accept or reject program transformations must be based on a well-defined program policy. This paper describes our experiences in designing a system to control dynamic program transformations.

KEY WORDS
Mobile code, Dynamic Optimization, Adaptive Optimization, Integrity, Program analysis, Computer security

1. Introduction

It has been shown that dynamic, adaptive optimization of programs during execution are an effective technique to provide an appreciable increase in program performance [1]. Dynamic program optimization transforms programs 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 the 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 [2]. These gains are achieved by allowing the computation to continue on one host while transforming it in parallel on another.

This paper defines dynamic program transformation to be any addition or deletion of instructions within a program while the program is running. Allowing phases of dynamic program transformation/optimization to occur in parallel on different hosts 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 [3], [4]. Before adopting such a distributed, dynamic, adaptive transformation technology, 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. Dynamic optimization performed in parallel by different hosts creates the situation where one host is executing code generated by another host. 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 have been applied to a program. The decision to accept or reject program transformations must be based on a well-defined policy for the program.

This paper presents our DOCTORS framework (for Distributed Online Control for Transformation and Optimization of Running Software). 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, (2) program transformation requests may be generated either

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.

by the server or by the client node, and (3) validation of transformation requests may occur on either the client node, the server, or on both the client node and server.

The remainder of the paper is organized as follows. In the next section, we provide relevant background material for this problem. Then, we present an overview of the framework for our proposed system. We then provide details on specification languages and system implementation within the framework. We then present some results and analysis of an implementation of our system. Lastly, we conclude with a discussion of related work and a summary of our contributions.

2. Background

There exist a number of techniques for and applications of dynamic software transformation. Examples of dynamically evolving software can be seen in systems that utilize just-in-time compilers (JITs) [5], dynamic optimizers [1], [2], dynamic translators [6], and dynamic code instrumentation systems [7]. The 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). This section introduces some of those techniques and applications in order to provide the reader with sufficient background to understand the issues associated with dynamic software transformation and program integrity.

All of the above techniques deal with program transformation, instrumentation, or profiling. They all involve modifying (either temporarily of permanently) instructions for computations or flow of control within a program. None of the above techniques provide control over how the program is transformed at runtime for the sake of security. A vast amount of research targets security issues related to non-evolving mobile code. Current techniques include computing checksums (e.g., HMAC) [8] over the object file of the software and digitally signing the software (e.g., RSA and DSA) [9], [10]. 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). Furthermore, this approach does not address the problem of control over how the program evolves or changes during execution. This paper presents techniques to accomplish control over how a program can change during execution based on security policy.

There exists a delicate balance between designing an automated system which is both flexible and robust against malicious code. The two opposing goals are to design a system with enough flexibility to permit program changes that are beneficial to the user, yet one that 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 [11], [12]. This is why many present-day methods for detecting previously categorized malicious code patterns rely on pattern matching techniques. 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.

3. Control Framework

For the purposes of this paper, the terms *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 [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].

Figure 1 presents a high-level view of an example of the type of 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 policy
- At least one *Client Node* that will execute the software and possibly request 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 a pool of nodes designed to assist the program transformation process – the Server Pool is treated as a single entity
- Perhaps one or more *Malicious Nodes* that may attempt to transform the program in a nefarious manner

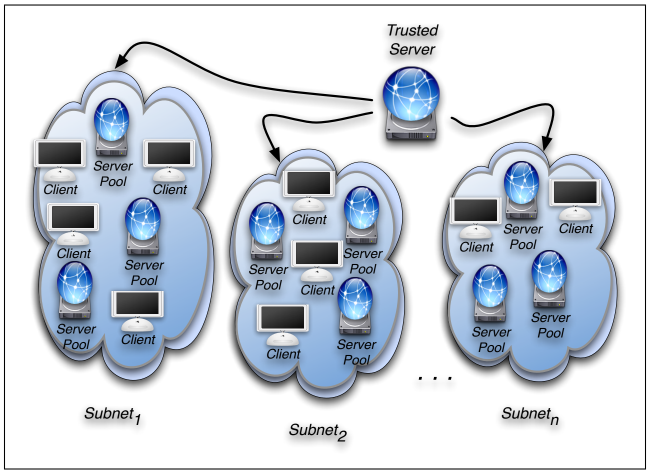

Fig. 1: Generalization of network operating environment.

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 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 function as the Server Pool.

Server Pool technology has been devised to provide reliable server services to networks [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.

3.1 Transformation Control

An example scenario giving the details of our system follows. The Trusted Server publishes 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 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 transformation(s) to be applied at a precise location 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.

3.2 Generic Framework

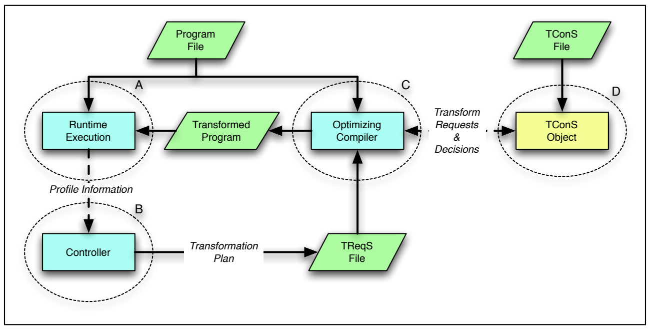

Fig. 2: Overview of controlled execution environment.

A generic view of the DOCTORS framework is shown in Figure 2. In this figure, a program file represents the executable instructions of a program. This file is run on the local client within the dashed circle labeled “A”. Dynamically generated profile information is provided to the controller, “B”. This information is used to generate a list of desired program transformations. This list is represented by the TReqS File. The TReqS File is consulted by the Optimizing Compiler, “C” and validated against the transformation specification represented by the TConS Object, “D”. Within the goals of the DOCTORS framework, the activities at each dashed circle of Figure 2 can take place at either a Client Node or on the Server Pool.

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).

4.1 The Transformation Request Language

Details on the grammar for the TReqS language can be found in Figure 3. The TReqS language is a simple language designed to record the type of transformation being requested and the location in the program where the transformation is to be applied. The language was designed to be human readable to facilitate empirical study and evaluation of the DOCTORS framework.

```
req:        <transform> [<req>]

transform:  <branElim>  | <cnstFold>
            <cnstProp>  | <copyProp>
            <cse>       | <deadCode>
            <gvn>       | <inline>
            <loopUnrol> | <noRHS>
            <simple>    | <tailRecur>


branElim:   BRANCH:<class>:<method>:<addr>

cnstFold:   CNST_FOLD:<class>:<method>:<addr>

cnstProp:   CNST_PROP:<class>:<method>:
                      <addr>:<const>:<use>

copyProp:   COPY_PROP:<class>:<method>:
                      <addr>:<copy>:<use>

cse:        CSE:<class>:<method>:<addr>

deadCode:   DEAD_CODE:<class>:<method>:<addr>

gvn:        GVN:<class>:<method>:<addr>

inline:     INLINE:<class>:<method>:<addr>

loopUnrol:  LOOP:<class>:<method>:<addr>

tailRecur:  TAIL:<class>:<method>:<addr>

simple:     SIMPLE:<class>:<method>:<addr>
```

Fig. 3: Simplified grammar for TReqS language.

The format of the TReqS is very straightforward, as shown in the following example:

```
COPY_PROP:MyClass:foobar:32:fooCopy:fooUse
CSE:OtherClass:foo:12
```

In the above example, two transformations have been requested. Individual fields within the request are separated by the colon character, ":". The first request is to perform copy propagation. The specific request is to propagate the value of fooCopy to fooUse at address 32 in method foobar of class MyClass. The second request is to perform common subexpression elimination. The request is to eliminate a common subexpression at address 12 in method foo of class OtherClass.

4.2 The Transformation Control Language

The TConS language is able to specify permitted and disallowed transformations within a program based on transformation type, location (e.g., program, class, method, and statement granularity), or parameters (e.g., variables, constant values, method names, etc.). Figure 4 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.

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

<method_rule> := /* epsilon */
| METHOD <method_name> {
    <transformation> }

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

<transformation_rules> := /* epsilon */
| /* <transformation specific
     rule productions> */

<class_name> := /* ID token */

<method_name> := /* ID token */

<addrange> := <addr> : <addr>

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

Fig. 4: Simplified grammar for the TConS language.

From Figure 4, we see that a TConS specification takes the general form:

```
TCONS CLASS "class_name" {
  METHOD "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 matching BEGIN and END tokens (left and right curly braces, respectively). Transformations can be specified on a class, method, or address level. The transformations that this system is currently 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
- Tail recursion elimination
- Loop unrolling

With one exception, the above transformations are those commonly employed in optimizing compilers and defined in [16]. 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.

The grammar was designed in a generic enough manner to be able to control many kinds of program transformation. The prototype implementation of the system was designed to handle only those program transformations as implemented by the JikesRVM, the basis of the prototype implementation and evaluation.

TConS files are generated by hand in the prototype implementation of the DOCTORS framework. The set of transformation controls available in DOCTORS are specific to program optimizations within the JikesRVM [1] system. Within the set of controlled program transformations, control can be specified at the class, method, and statement level for individual variables, or for entire types of program transformations. The default behavior of the policy can be defined to be restrictive (prevents all transformations unless specifically permitted within the TConS file) or permissive (permits all transformations unless specifically prevented within the TConS file).

Within the DOCTORS framework, there is no guarantee that a given transformation will be applied to a program if the policy in the TCONS file indicates that the transformation is permitted. The only guarantee in this example is that if the system requests to apply a permitted transformation during program execution, then the system will be permitted to apply that program transformation. If a transformation is not identified as a permitted transformation by the TConS file, then the DOCTORS framework will prevent the system from applying that transformation during program execution.

4.3 Example Control Situation

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 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.

Figure 5 shows a simple example Java program. In this example, the designer of the system wants to protect the value of the variable k as defined at address 32 in the method main(). Figure 6 shows the backwards slice starting at the point in the program where the value of k is defined. This backwards slice contains all statements that affect the value of k at address 32. The slice is defined thus:

- At address 32 in main(), the value assigned to k is returned from the call to subtract().
- At address 16 in subtract() the value returned is defined by the expression $c \quad d$.
- At addresses 8 and 12 in subtract(), the values of c and d are defined by variables a and b, respectively.
- At address 0 in subtract(), the values for the formal parameters a, and b, are defined by the values of the actual parameters w and z in the call site at address 32 in main().
- At address 12 in main() the value of w is defined as 5.
- At address 24 in main() the value of z is defined by the value of x.
- At address 16 of main(), the value of x is defined as 10.

```
   public class foo {
     public static void
 0   main(String args[]){
 4     int j, k;
 8     int w, x, y, z;
12     w = 5;
16     x = 10;
20     y = w;
24     z = x;
28     j = subtract(y, x);
32     k = 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; } }
```

Fig. 5: Example Java program.

The TConS specification shown in Figure 7 defines policies for methods `main` and `subtract` of class `foo`. The default return value for this policy is false, making this specification a restrictive policy – all transformations are

[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) [17].

```
  public class foo {
    public static void
 0  main(String args[]){
 4    int j, k;
 8    int w, x, y, z;
12    w = 5;
16    x = 10;
24    z = x;
32    k = 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; } }
```

Fig. 6: Example of a backwards slice in method main().

prevented except those explicitly permitted by the policy. 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.

```
TCONS CLASS foo {
 METHOD main {
  COPY_PROP {
   /* At address 0-32, may prop. x to z */
   address(0:32, x(z)); }
   CONSTANT_PROP {
    /* At address 0-32, may prop. to y, may
       prop. to x & z if constant is 10 */
    address(0:32, z(10), y(),
            x(10), w(5)); } }
 METHOD subtract {
  NO_RHS {
   /* At address 0-16, may not apply any
      transforms that affect the value */
   address(0:16); } } }
```

Fig. 7: Example TConS specification to protect slice of k.

Note that in this instance, the slice contains every statement in the program except the statements at addresses 20 and 28 in method main(). Figure 7 shows a TConS file to protect the slice given in Figure 6. This TConS file defines the only permissible transformations within the method main() to be the following: (1) Copy propagation, if the value of x is being propagated to z, and (2) Constant propagation, if the value being propagated to x or z is ten, if the value being propagated to w is 5, or any constant value may be propagated to y. Within the method subtract(), no transformations are permitted that change the values of the right-hand side of any assignment statement.

The TConS file in Figure 7 protects the computations involved in the definition of k at address 32 in method main(). This TConS file achieves this protection by prohibiting all transformations that would alter the original computation of the value of k in the program. In essence, the TConS file prevents transformations that change the semantics of any instructions in the program that appear in the backwards slice for the value of k at address 32 in method main().

5. Evaluation

This section presents the research questions designed to guide the empirical study. The design and details of the study are then given and discussed. Lastly, the specifics of the transformation infrastructure and simulation environment are presented.

5.1 Research Questions

The specific questions that were used to guide the design of the empirical study are the following: 1) Correctness: Does the DOCTORS framework achieve the goal of transformation control correctly, and without adverse side effects? 2) Cost: How expensive is it to control dynamic program transformation (i.e., how much overhead is introduced)?

5.2 Transformation Infrastructure

Our framework was implemented within the JikesRVM [1] environment. As such, transformation controls were selected based on program transformations that JikesRVM actually performs.

Figure 8 presents an instantiation of the DOCTORS framework as implemented in JikesRVM with the addition of the TConS functionality (JikesRVM$^+$). The numbered circles (1 through 7) in the figure represent various points of activity in the RVM that are studied. At the start of program execution, the **Baseline Compiler** compiles relocatable **Java class files** to **Executable Code** for the target host machine. This initial native version of the Java program has a limited set of very basic optimizations applied beyond those performed by the Java compiler used to produce the initial Java bytecode. A **Controller** monitors the program during execution and makes decisions for dynamic re-optimization based on program **Runtime Execution** profile information. From this profile information, the controller puts together an optimization plan that the **Optimizing Compiler** uses to re-optimize portions of the Java program. This re-optimized **Executable Code** replaces the current version of the program in execution.

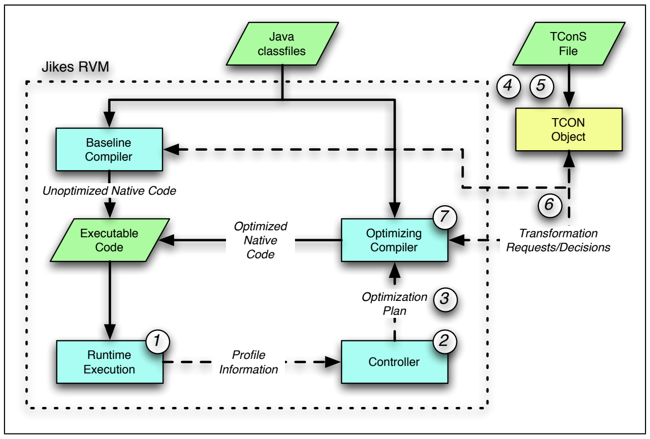

Fig. 8: Overview of modified JikesRVM.

Table 1: Benchmark Programs Used in DOCTORS Study.

Benchmark Name	Size (KB)	# of Classes	Program Description
check	39.43	17	simple program to test the VM
compress	17.40	12	LZW compression/decompression engine
jess	10.40	151	puzzle solving Java Expert Shell
raytrace	5.57	25	light path modeling engine
javac	5479.94	771	JDK 1.0.2 compiler
mpegaudio	117.37	55	commercial audio file decompression
mtrt	0.84	1	variant of **raytrace**
jack	127.82	57	Java parser generator

5.3 Subject Programs

Table 1 presents the benchmark programs used in this study to test the DOCTORS framework. The benchmark name is given, along with the total size of the benchmark (with no compression, to include all class files and data files), the total number of classes, and a brief description of the computation performed by the benchmark program. Many subject programs used in this study are composed of multiple Java class files. To measure the total size of the subject program, the size of all class files must be included. Table 1 shows program size without compression to accurately present the total size of each subject program. After compression, file size will vary, depending up the compression algorithm selected.

5.4 TConS/TReqS Grammar Details

The grammar for the TConS language [18] was written for, and interpreted by javacc (a Java Compiler Compiler). The output of javacc is a parser, which when fed a TConS file as input, creates a TCON object to handle transformation requests and responses according to policy specified within a TConS file. This object was added to the JikesRVM to facilitate transformation control.

The TReqS language is much less complex than the TConS language. The only information required to represent a transformation request in a TReqS is the type of transformation and any possible operands or values to the transformation at a single location within the program. A TConS must include information about the type, location, operands, and permissibility of a transformation throughout the entire program. As a result of this simplicity, the TReqS language can be processed without generating complex parse trees. The object that handles TReqS queries simply parses the request one field at a time, retrieving the specifics of the request to consult the TConS regarding the permissibility of that request.

Table 2: Names and Descriptions of TConS Files.

Name	Description
0	Permits 0% of original transformations
10	Permits 10% of original transformations
25	Permits 25% of original transformations
50	Permits 50% of original transformations
75	Permits 75% of original transformations
90	Permits 50% of original transformations
100	Permits 100% of original transformations
mix	Manual Introspective eXperiment

To generate the TConS files for each benchmark, the benchmark was first executed without transformation controls. During the control-free execution, transformation requests were recorded. Each benchmark program was run in the DOCTORS framework in this manner, permitting all transformation requests made by the optimizing system. These requests were recorded in a TReqS file to generate a baseline of transformation requests for each benchmark program. This baseline TReqS was then used as a pool of requests that would be individually designated as either permitted or prohibited in that program for evaluation. Many transformation requests were repeated in the baseline TReqS because of the way JikesRVM implements dynamic recompilation. Thus, the baseline TReqS was sorted and redundant requests were removed. The total number of resulting requests in the TReqS for each benchmark was counted. This total number of requests was then used to calculate absolute thresholds for 10, 25, 50, 75, and 90% of the total number of transformation requests. Transformations were chosen at random from the TReqS pool to be added to the TConS file until each threshold number of transformations was met. The randomly chosen transformation requests were identified as permitted transformations and recorded in the TConS file. Thus, TConS files were created that permitted 10, 25, 50, 75, and 90% of the baseline transformation requests. The names and descriptions of the TConS files are given in Table 2. A security-minded TConS file was also generated by hand, through examination of program source code and class files. Specific regions and values in the benchmark program were chosen for protection. These chosen regions and values were protected from transformation in a TConS file named “mix” for each benchmark program.

5.5 JikesRVM Revisions

Several modifications were made to the JikesRVM in order to provide proof-of-concept for this work and to collect empirical data that measures the effect that this technique has on dynamic program transformation. Code was added to JikesRVM to accomplish two goals: 1) to enact transformation control, and 2) to collect measurements (time and transformation count) on various aspects of the system.

The largest modification to the RVM was the addition of the TCON object to the main virtual machine object (VM.java). For this modification, the TCON object was added as another field of the VM object. Placing the TCON object within the main object of the virtual machine facilitated access to the TCON object for transformation queries throughout the virtual machine. Thus, at every point in the dynamic compiler where a transformation is about to be applied to a program, the TCON object is consulted before applying the transformation.

The following JikesRVM source code files were modified to add TCON object query and transformation control:

- OPT_LoopUnrolling.java
- OPT_TailRecursionElimination.java
- OPT_Simple.java
- OPT_GlobalCSE.java
- OPT_GlobalValueNumber.java
- OPT_GlobalValueNumberState.java
- OPT_RedundantBranchElimination.java
- OPT_InlineDecision.java
- OPT_Inliner.java
- OPT_LocalCSE.java
- OPT_LocalConstantProp.java
- OPT_LocalCopyProp.java
- OPT_Simplifier.java

In essence, immediately before the point in each of the above source code files at which the respective transformation is about to be applied, a call was inserted to query the TCON object. The call provides specific information pertaining to the kind of transformation to be applied, the location within the program (i.e., class and method name, and bytecode address within the method), and the operands (if any) involved in the transformation. The TCON object consults its internal representation of the TConS file and provides a response to the RVM transformation request. Based on the nature of the response, the transformation is either applied or ignored.

5.6 Results

We present charts of the measurements taken during each benchmark execution within the DOCTORS framework to answer the research questions pertaining to cost. Data reported in this section is for measurements taken on a single machine. Where normalized data is reported, the data is normalized to the base case (i.e., the case where all dynamic program transformations are permitted).

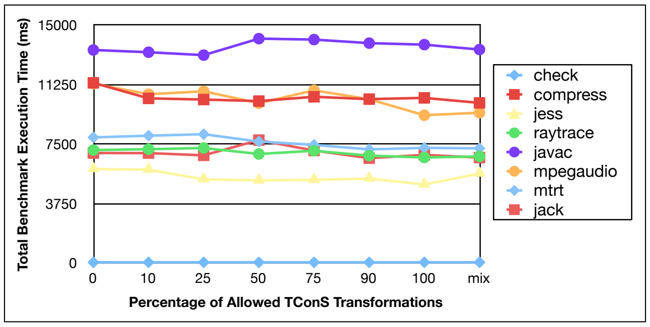

Fig. 9: Total Execution Time per Percentage of Allowed TConS Transformations on a Single Client.

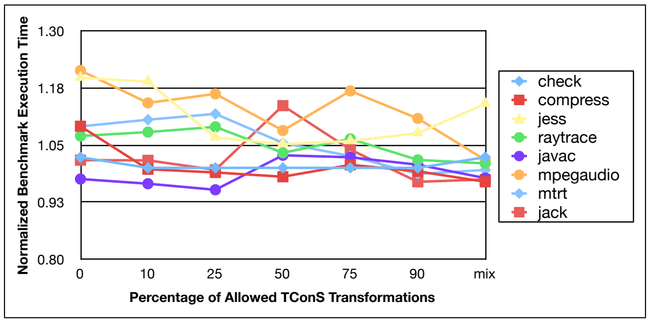

Fig. 10: Normalized Execution Time per Percentage of Allowed TConS Transformations on a Single Client.

Execution times for benchmark programs run within the DOCTORS framework with different percentages of TConS controlled transformations are given in Figures 9 and 10. Figure 9 shows the total time required for each benchmark within the DOCTORS framework. Normalized benchmark execution times are presented in Figure 10.

As can be seen by Figure 10, the DOCTORS framework introduces an overhead in benchmark execution time with an upper bound of 20%. This time increase results from several factors: 1) It takes time to read the TConS file and create the TCON object, 2) The optimizing compiler spends time on program analysis for transformations that will not be applied to the program because some transformations are not-permitted by the TConS, 3) The program analysis is repeated throughout the benchmark program's execution since the RVM continuously tries to recompile and optimize the benchmark program, and 4) Potential performance-improving program transformations are being prevented from enabling performance gains, thus execution time does not improve (i.e., decrease). While there is a slight trend towards reduction in execution time as more transformations are permitted, the data is too varied to make this claim with confidence.

5.7 Summary of Results

We now summarize the evaluation results with respect to the research questions presented in Section 5.1.

5.7.1 Correctness: Proving correctness of individual program transformations and combinations of program transformations is a very difficult and open problem [19], [20]. Thus, correctness of DOCTORS was measured by executing the controlled transformation version of a program and comparing the output with the original, unrestricted program execution. All of the benchmark programs belong to the well known SPEC benchmark suite which includes test data and expected results. In all cases of all controlled transformation executions (i.e., 0 - 100, and mix TConS files) for all benchmark programs, the execution results matched the unrestricted runs of the programs. Thus, while correctness is not formally proven, all program runs proved correct, a strong indication of correctness.

To measure DOCTORS' ability to correctly prevent/permit program transformations according to TConS policy, program transformation traces and TConS files where studied. Based on the studies of the program transformation transcripts and TConS files, the DOCTORS framework correctly prevents all instances of transformations that are identified as not permitted in a TConS file. Transformations that are identified as permissible within the TConS are correctly granted upon request from the RVM.

5.7.2 Cost: From the subject programs in this study, the DOCTORS framework introduces an overhead with an upper bound of 20% in terms of program execution time over all benchmark programs studied, based on results reported in Figure 10. While there appears to be a small correlation between an increase in the number of allowed transformations and a decrease program execution time (See Figures 9 and 10), there is too much variance in the data to make this claim with confidence.

Additional space, ranging from 128 bytes to 111MB (for both TConS file and TCON object) is required to store the TConS file on disk and to store the TCON object in memory during program execution. Space requirements can be reduced through more efficient file encoding and through an extensive rewrite of the prototype. Designing TConS files that are similar in form to the "mix" TConS files in this study will help in addressing the issue of space required for the TConS file and TCON object.

6. Related Work

Cai et al. provide a Hoare-logic-like framework to verify self-modifying machine code [21]. This is accomplished by treating code not as immutable text, but rather as regular data. This framework can verify program partial-correctness. Myreen has also produced a system to address program correctness, particularly in a just-in-time compilation environment [22]. The system presented by Myreen also utilizes a Hoare logic to describe pre and post conditions for program verification and transformation.

Both of these systems are designed to verify the correctness of some given program. The key difference between these systems and our framework is that our framework is concerned with controlling what transformations can be applied to a program and where, while the work of both Cai et al. and Myreen target program correctness. There is a subtle difference between these goals. Our framework may permit transformations that would change the semantics of the program (which would be considered to incorrect to an optimizing compiler, compilers are designed to preserve program semantics).

Necula's work on Proof-Carrying Code (PCC) [23], and other's extensions to PCC [24] 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. PCC will not guarantee that 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 addresses 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.

Pleszkoch and Linger describe a technique which they term function extraction to automatically extract program behavior from source code in order to detect malicious behavior [11]. This technique employs a function-theoretic model which treats programs as mathematical functions that 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 for large-scale software systems.

This differs from our work in that function extraction searches for known bad behaviors in programs.

Voss and Eigemann present a method of automated, de-coupled, adaptive program transformation in [2]. In this work, the authors de-couple the phases of dynamic adaptive transformation to parallelize the process. In this way, the authors improve upon the performance gains of dynamic adaptive transformation by allowing the local host to continue computation while another host is performing program optimization. When new versions of the program are available, they are replaced on the local host. We extend this work to allow hosts to control how a program is transformed based on 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 operate in open network environments.

7. Conclusions

For the programs and transformations in this study, the DOCTORS framework correctly controls dynamic program transformations according to policy specified within a TConS file. While the proof-of-concept system has some adverse effect on runtime performance (i.e., additional computation time and storage), the goals for correctness and transformation validation were achieved.

The results in this study were in accordance with expectations. The DOCTORS framework has additional cost associated with using the system. However, the costs of the framework can be managed to maintain reasonable levels through careful planning of TConS file creation. The first criterion one must take into consideration when implementing the DOCTORS framework are the following: 1) Is dynamic program transformation required for the computation in the network? and 2) Is control over dynamic program transformation desirable?

If dynamic transformation is not being utilized, or if control of program transformation is not required, then the DOCTORS framework is neither required nor beneficial. If control of dynamic program transformation is desirable, then the DOCTORS framework is recommended.

References

[1] 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, pp. 111–129.

[2] 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.

[3] 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*, 2002, pp. 141–148.

[4] 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.

[5] 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*, June 2000.

[6] K. Ebcioğlu 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, 1997.

[7] 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.

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

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

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

[11] 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.

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

[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. A. Fecko, U. C. Kozat, S. Samtani, M. Ümit Uyar, and I. Hökelek, "Reliable and dynamic access to service in battlefield ad hoc networks," in *Proceedings – IEEE Military Communications Conference MILCOM*. IEEE, Nov. 2004.

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

[17] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, 1990.

[18] M. Jochen, A. Anteneh, L. Pollock, and L. Marvel, "Enabling control over adaptive program transformation for dynamically evolving mobile software validation," in *Software Engineering for Secure Systems (SESS–05)*, May 2005.

[19] S. L. Gerhart, "Correctness-preserving program transformations," in *Proceedings of the 2nd ACM Symposium on Principles of Programming Languages*, 1975.

[20] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, 2004.

[21] H. Cai, Z. Shao, and A. Vaynberg, "Certified self-modifying code," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2007.

[22] M. O. Myreen, "Verified just-in-time compiler on x86," in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2010, pp. 107–118.

[23] G. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Jan. 1997.

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