TRUSTED COMPONENT SHARING BY RUNTIME TEST AND IMMUNIZATION FOR SURVIVABLE DISTRIBUTED SYSTEMS

Joon S. Park[1], Pratheep Chandramohan[2], Ganesh Devarajan[3], and Joseph Giordano[4]

[1,2,3]Laboratory for Applied Information Security Technology (LAIST), School of Information Studies, Syracuse University; [4]Information Directorate, Air Force Research Laboratory

Abstract: As information systems became ever more complex and the interdependence of these systems increase, the survivability picture became more and more complicated. The need for survivability is most pressing for mission-critical systems, especially when they are integrated with other COTS products or services. When components are exported from a remote system to a local system under different administration and deployed in different environments, we cannot guarantee the proper execution of those remote components in the currently working environment. Therefore, in the runtime, we should consider the component failures (in particular, remote components) that may either occur genuinely due to poor implementation or the failures that occurred during the integration with other components in the system. In this paper, we introduce a generic architecture and mechanisms for dynamic component-failure detection and immunization for survivable distributed systems. We have also developed a prototype system based on our approaches as a proof of our ideas.

Keywords: Component Immunization; Recovery; Survivability.

1. INTRODUCTION

Although advanced technologies and system architectures improve the capability of today's systems, we cannot completely avoid threats to them. This becomes more serious when the systems are integrated with

Commercial Off-the-Shelf (COTS) products and services, which usually have both known and unknown flaws that may cause unexpected problems and that can be exploited by attackers to disrupt mission-critical services. Usually, organizations including the Department of Defense (DoD) use COTS systems and services to provide office productivity, Internet services, and database services, and they tailor these systems and services to satisfy their specific requirements. Using COTS systems and services as much as possible is a cost-effective strategy, but such systems—even when tailored to the specific needs of the implementing organization—also inherit the flaws and weaknesses from the specific COTS products and services used. Traditional approaches for ensuring survivability do not meet the challenges of providing assured survivability in systems that must rely on commercial services and products in a distributed computing environment[31, 29, 30].

The damage caused by cyber attacks, system failures, or accidents, and whether a system can recover from this damage, will determine the survivability characteristics of a system. A survivability strategy can be set up in three steps: protection, detection and response, and recovery[21, 16, 18]. To make a system survivable, it is the mission of the system, rather than the components of the system, to survive. This implies that the designer or assessor should define a set of critical services of the system to fulfill the mission. In other words, they must understand what services should be survivable for the mission and what functions of which components in the system should continue to support the system's mission[25].

2. DEFINITION OF SURVIVABILITY

The definitions of survivability have been introduced by previous researchers[20, 23]. In this paper, we define survivability as the capability of an entity to continue its mission even in the presence of damage to the entity. An entity ranges from a single component (object), with its mission in a distributed computing environment, to an information system that consists of many components to support the overall mission. An entity may support multiple missions. Damage can be caused by internal or external factors such as attacks, failures, or accidents. If the damage suspends the entity's mission, we call it *critical damage* (CD), and if it affects overall capability, but the mission can still continue, we call it *partial damage* (PD). Since we believe survivability is a mission-oriented capability, there are basically three abstract states of the system: normal, degraded, and suspended. A system is in the normal state (S_0) when it is running with full capability. It is in the degraded state (S_1) when it is running with limited capability because of PD, which does not suspend the overall mission. Finally, the system is in the suspended state (S_2) when it cannot continue its mission because of CD. When partial recovery (PR) occurs to an infected component, only the mission-related service is recovered, so the service is still in a degraded mode with limited capacity. When there is a total recovery (TR) such as that resulting from component substitution, service is provided at full capacity. As understood intuitively, PR and TR on S_0, PD and PR on S_1, and PD and CD on S_2 do not change their current states. From the survivability point of view, we may put up with partial damages (PD) on the system as long as the mission carries on. We may simply insulate the damaged components from others instead of recovering them, although the performance of the overall system may degrade. However, if the damage is so critical that the system cannot continue its mission, we must recover the damaged components as soon as possible in order to continue the mission. We describe the concept of survivability using a finite state machine. Abstractly, we can consider the damages and recovery actions as inputs to a survivable entity. Furthermore, we can classify the outputs of the entity into two abstract cases, one for the outputs when the mission performed (M) successfully, and one for the outputs when the mission failed (F). This generates Table 1, which shows the transitions and outputs for each pair consisting of a state and an input. Based on this table, we generate a finite state machine in Figure 1.

Table 1. State Table for Survivable Systems

State (S)	Transition Function (f)				Output Function (g)			
	Next State				Output (O)			
	Input (I)				Input (I)			
	PD	CD	PR	TR	PD	CD	PR	TR
S_0	S_1	S_2	S_0	S_0	M	F	M	M
S_1	S_1	S_2	S_1	S_0	M	F	M	M
S_2	S_2	S_2	S_1	S_0	F	F	M	M

The finite-state machine M = (S, I., O, f. g, s_0) consists of a finite set S of states (where S_0 is an initial state), a finite input alphabet I, a finite output alphabet O, a transition function f that assigns each state and input pair to a new state, and an output function g that assigns each state and input pair to an output. In this state diagram, we have three states (normal state (S_0), degraded state (S_1), and suspended state (S_2)), four types of inputs (partial damage (PD), critical damage (CD), partial recovery (PR), and total recovery (TR)), and two outputs (when mission performed (M), and when mission failed (F)).

To continue the mission, the system must stay in either S_0 or S_1. Some strict missions do not allow the critical components to stay even one moment in the suspended state (S_2) until the mission is completed. However, in reality, we believe most missions may allow critical components to stay in the suspended state (S_2) for a moment until they are recovered and the state is changed to the degraded state (S_1) or normal state (S_0).

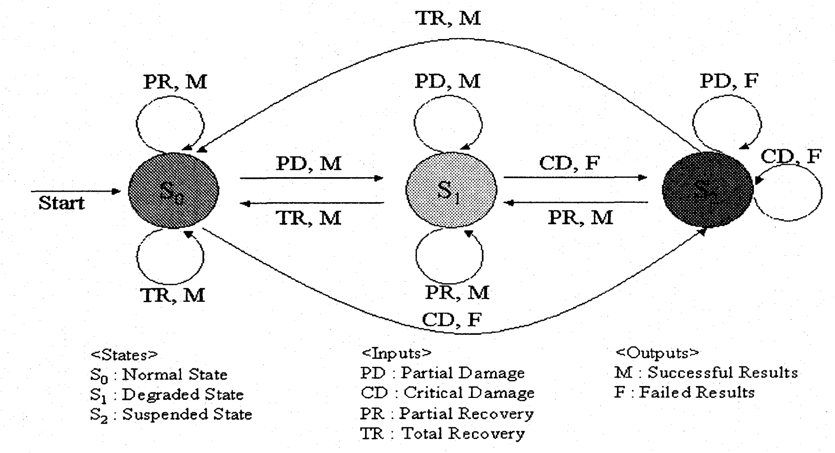


Figure 1. Abstract State Diagram for Survivable Systems

We could decompose S_1 and S_2 into sub-states to represent detailed transitions based on the actual missions and applications described in[20]. In this paper, however, we introduce a generic approach to describe the concept of survivability with the abstract inputs, states, and outputs. We believe the three states (S_0, S_1, and S_2), the four kinds of inputs (PD, CD, PR, TR), and the two kinds of outputs (M, F) can represent the state transitions of a survivable entity based on our mission-oriented survivability definition.

3. RELATED WORK

3.1 Black-box and white-box testing

Currently, existing technologies for identifying faulty components are more or less static in nature. One of those approaches employs black box testing for the components. In this technique, the behavioral specification[2] is provided for the component to be tested in the target system. The main disadvantage of this technique is the specifications should cover all the detailed visible behavior of the components, which is impractical in many situations. Another approach employs a source-code analysis, which depends on the availability of source code of the components. Software testability analysis[35] employs a white-box testing technique, which determines the locations in the component where a failure is likely to occur. Unlike black box testing, white box testing allows the tester to see the inner details of the component and later help him to create appropriate test data. Yet another approach is software component dependability assessment[36], a modification to testability analysis where each component is tested thoroughly. These techniques are possible only when the source code of the components is available.

3.2 Fault injection

In the past[19] we have employed a simple behavioral specification utilizing execution-based evaluation. Their approach combines software fault injection[1, 24, 33, 34] at component interfaces and machine learning techniques to: (1) identify problematic COTS components; and (2) understand these components' anomalous behavior. In their approach of isolating problematic COTS components, they created wrappers and introduced them into the system under different analysis stages to uniquely identify the failed components and to gather information on the circumstances that surround the anomalous component behavior. Finally, they preprocess the collected data and apply selective machine learning algorithms to generate a finite state machine for better understanding and increasing the robustness of the faulty components. In other research[7] the authors have developed a dynamic problem determination framework for a large J2EE platform, employing a fault detection approach based on data clustering mechanisms to identify faulty components. This research also employed a fault injection technique to analyze how the system behaves under injected faults.

3.3 Bytecode instrumentation

Performing fault injection analysis and providing immunization to the components either by rewriting the existing code or by creating additional wrappers is a non-trivial task when the source code for the component is not readily available. Source code may not be available at all when we are dealing with COTS components and externally administered components downloaded dynamically in runtime at local machine. This is an issue that needs to be addressed before proceeding further. Providing immunization and performing fault injection at the component interfaces require modification of the component code; however, we assume that the source code is *not* available in a large disturbed application. Instead, we provide the immunization to the runtime code (e.g., JAVA Bytecode) by extending the code instrumentation technique[5, 6, 8, 10, 15, 17]. Instrumentation techniques have previously been used for debugging purposes; to evaluate and compare the performance of different software or hardware implementations such as branch prediction, cache replacement, and instruction scheduling; and in support of profile-driven optimizations[3, 9, 11, 22].

4. RUNTIME COMPONENT TEST AND IMMUNIZATION

4.1 Generic system architecture

Figure 2 shows the generic architecture of our component failure detection and immunization system. It consists of a Monitoring Agent, an Immunization Agent, and a Knowledge Base. The monitoring agent is further divided into two subsystems: the fault injection subsystem and fault detection subsystem. Before a component is run on a host (especially a mobile component downloaded from another machine under different administration), the fault injection subsystem injects faults into the component, while the fault detection system analyzes component behavior in response to the injected faults. The component's internal structure information, such as method interface, arguments, local variables, etc., is accessible in runtime; thus, this information can be used in the dynamic component analysis.

If there is no abnormal behavior, the monitoring agent allows the component to run in the local machine. For the performance reason, we can finish this analysis with the local components and fix the failures in the source codes before the operation starts (if the source codes are available). However, this is not possible for the remote components because their source codes are usually not available to the local machines. When the monitoring agent detects abnormal behaviors in the mobile component through the fault injection analysis, the fault detection subsystem identifies the reason for failure and informs the immunization agent to immunize the faulty component accordingly.

The immunization agent builds and deploys immunized components to the target system. The immunization agent possesses a knowledge base that consists of a list of procedures for how to provide immunization for component failures. The immunization agent provides immunization and increases the survivability of the faulty components. Basically, there are two options to increase the survivability of the vulnerable components and to make it more robust[12]: (1) inform the vendor of the software problems and wait for a patch; or (2) immunize the components with wrappers or instrument the faulty methods with updated and modified methods for more robust behavior[4, 26]. The first technique is not feasible for dynamic runtime recovery from errors; consequently, we have adopted the second approach to provide immunization and increase the survivability of vulnerable components.

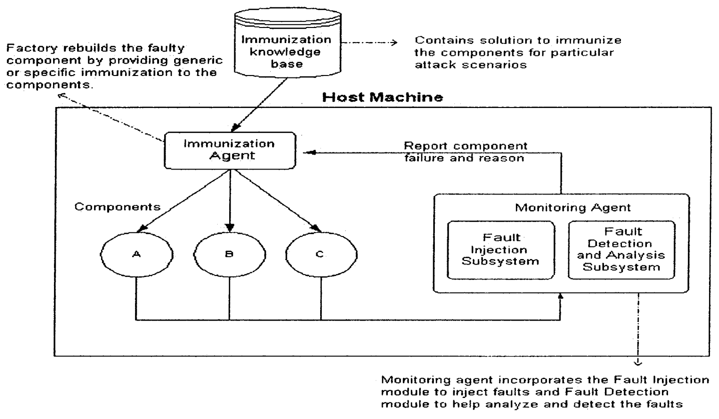

Factory rebuilds the faulty component by providing generic or specific immunization to the components.
Immunization knowledge base
Contains solution to immunize the components for particular attack scenarios
Host Machine
Immunization Agent
Report component failure and reason
Components
A
B
C
Monitoring Agent
Fault Injection Subsystem
Fault Detection and Analysis Subsystem
Monitoring agent incorporates the Fault Injection module to inject faults and Fault Detection module to help analyze and detect the faults

Figure 2. Component Failure Detection and Immunization

4.2 The strategy

Figure 3 summarizes the steps involved in the entire process of detecting and immunizing faulty components. When we download a component from the remote location we perform the first test to determine if there are any dependent components. If so, we also download the dependent component. The component that is downloaded is an executable file for which we don't have the source code. By using an additional tool in runtime (e.g. Jikes BT[15] for JAVA bytecode), however, we can determine most of the intricate structure details of the component that we have downloaded. The test as well determines the structure of the code (including the data flow and the interdependencies of the functions inside the component) that is required to do a runtime test in the local environment. Then, we go into the next phase of monitoring the component performance.

In the next phase we inject the faults and observe the performance of the component. The fault injection module injects test inputs (faults) and analyses the behavior of the component. Different machines (or applications) may have different fault injection modules based on their test criteria. For instance, one module may test internal failures, while another may test the robustness against cyber attacks. After the test inputs are injected we collect evidences and reasons for the failures, specific methods, classes that are affected. If there are any failures detected we check if we can provide some immunization to that failure from the knowledge base that we have built and updated regularly. If we have a specific solution for the failure we provide it from the knowledgebase, otherwise we provide it a generic immunization[27, 28]. After the immunization is done we send the immunized component to the monitoring phase again. Now if the component is not having any problem we go to the next phase where we see if all the fault injections are performed and the component is functioning without any problem then our goal is achieved. However, if there is any problem in the monitoring stage after the immunization we may simply drop the component off.

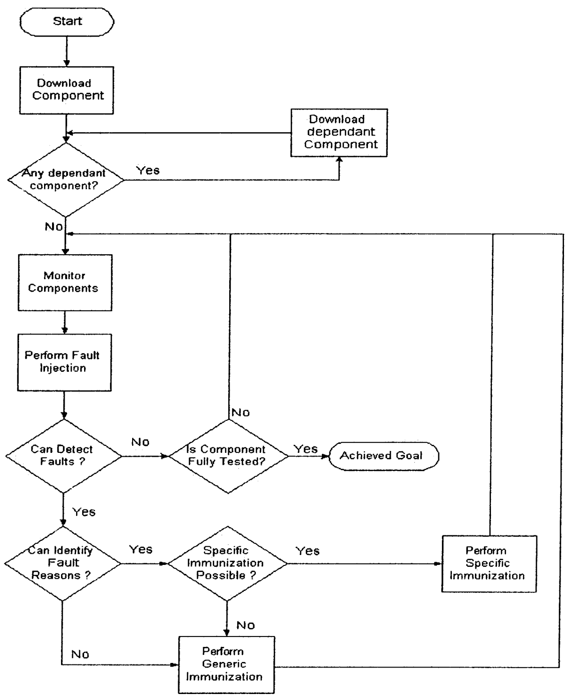

Start
Download Component
Download dependant Component
Any dependant component?
Yes
No
Monitor Components
Perform Fault Injection
Can Detect Faults ?
No
Is Component Fully Tested?
Yes
Achieved Goal
No
Yes
Can Identify Fault Reasons ?
Yes
Specific Immunization Possible ?
Yes
Perform Specific Immunization
No
No
Perform Generic Immunization

Figure 3. Strategy for Identifying Component Failures and Immunization

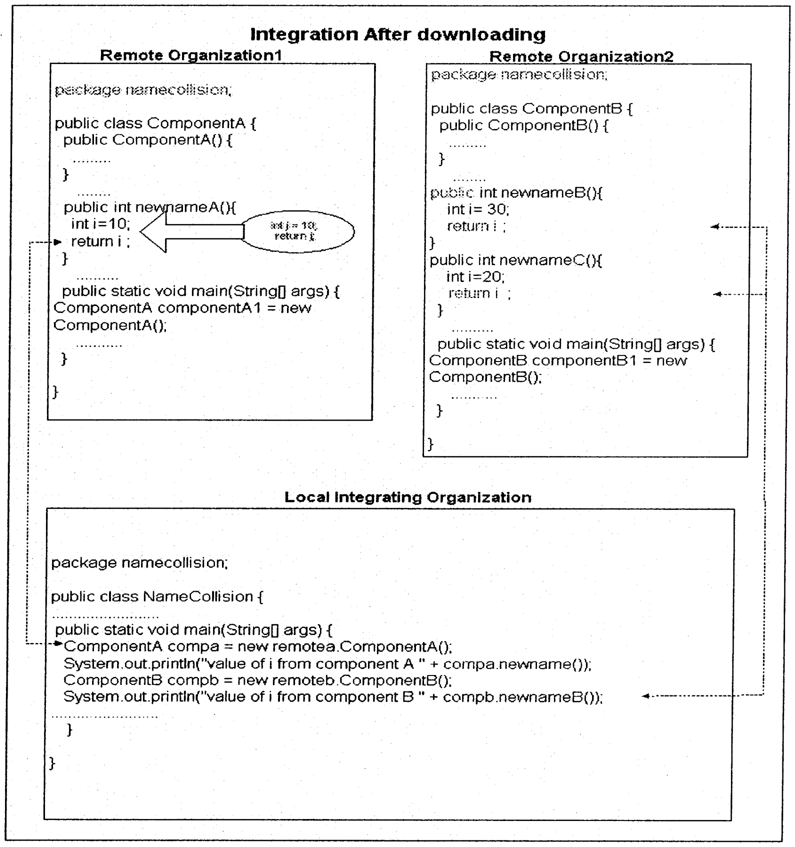
We can provide component immunization in runtime by either class-level modifications or method-level modifications. By class-level modification the references to the original class definitions are replaced by another subclass of the original class. By method-level modification, we modify some of the runtime (executable) code in the original method by adding new runtime code (i.e. Java bytecode in our implementation) or deleting some runtime code or both at the same time. The latter provides more flexibility to build more powerful immunized class. At the same time method level modification is more difficult to implement than class level modifications because it involves direct modification of already existing Java bytecode whereas the class-level modification just involves rearranging references in the class file. The main advantage of using method level instrumentation techniques is that all the modifications are transparent to other components, which make calls to the modified components because the semantics and syntax are still maintained after modifications.

5. PROTOTYPE DEVELOPMENT

Although the detailed techniques for component-failure detection and immunization are slightly different based on the programming languages, applications, and local policies, we believe our approach is applicable to most of distributed systems, which require survivability. We focus on the component failure scenarios here, but we believe our approach can be extended with cyber attacks. In our experiment, we detect component failures such as naming collisions, infinite loops, multi-threading problems, and array out-of-bound problems, and successfully immunized them in runtime so that the component's service can continue in a reliable manner. In the following description, we mainly concentrate on the problems of naming collisions because they cannot be rectified in the programming time and this particular paper has a space constraint. The other problems might be avoided when the programmer takes extra care during programming. However, we still need to check those problems in a remote component during runtime under a strict component-sharing policy.

5.1 Detection and immunization of naming collisions

When we perform tests for a local component, naming collision across other spaces cannot be detected. However, when we perform the test after the component is downloaded from a remote machine and integrated with local components there can be naming collisions occurring. There are possibilities that two or more components, which are being integrated together, might have the same variable name or even within the same component the same variable name can be used in different contexts. When the client program tries to access these variables there are possibilities that it might get the wrong value.



```
package namecollision;

public class ComponentA {
 public ComponentA() {
  .........
 }
  ........
 public int newnameA(){
  int i=10;
  return i ;
 }
  .........
 public static void main(String[] args) {
ComponentA componentA1 = new
ComponentA();
  ..........
 }
}
```

```
package namecollision;

public class ComponentB {
 public ComponentB() {
  .........
 }
  ........
public int newnameB(){
  int i= 30;
  return i ;
}
public int newnameC(){
  int i=20;
  return i ;
 }
  .........
 public static void main(String[] args) {
ComponentB componentB1 = new
ComponentB();
  ..........
 }
}
```

```
package namecollision;

public class NameCollision {
........................
public static void main(String[] args) {
 ComponentA compa = new remotea.ComponentA();
  System.out.println("value of i from component A " + compa.newname());
  ComponentB compb = new remoteb.ComponentB();
  System.out.println("value of i from component B " + compb.newnameB());
........................
 }
}
```

Figure 4. Detection and Immunization of Naming Collision

The downloaded component's internal structure information such as method interface, arguments, local variables, etc. is collected in runtime after analyzing the Java bytecode. Using the structure information and fault injection module, the local machine performs a fault injection test to determine all the return values in the component. This enables the local machine to figure out the architecture of the component and then to decide, which are all the function values required. Once the functions are selected the component is passed into the naming collision test stub where we test if there is any other component with the same variable name returning the value. If the testing says there are no variables with the same variable name then integration is taken to the next level.

Now if there are variables with the same variable name from different component then our immunization code for this scenario will be creating an instance for the remote class. Using this instance we access the method name and through that we access the variable value(e.g. compa.newnameA()). The renaming process to avoid naming collision is to be performed mainly when we convert the private function to public function. The original source code writer's intension would have been that the function was a private function its scope is well defined and hence he can reuse the variable name. If there is a private function then this will not affect our processing as that variable it limited to the scope of that class. However, if there are two variables from the same component with the same name then we can go about changing the name as per the naming convention so that it becomes easier for the programmer who is working with the source code generated from the bytecode to differentiate from the other common variable named item.

The main advantage of this system is that we can get access to the variables which where initially not possible to access and then by renaming them we are able to distinguish between the two similarly name variable. This as well helps in the optimal code re-usage. In reality, performing instrumentation is a non-trivial task because it involves precision handling of instructions. In most of the cases the instrumentation requires dealing with intermediate-level code (e.g., Java bytecode) or low-level code (e.g., Assembly), which requires ultra care when modifying these kinds of code. Basically our principle can be applied to more complex problems but the complexity of the immunization code increases quite considerably when dealing with complex problems. An important point should be noted here that it is not always possible to apply immunization by changing the code (Java bytecode in this case). In some cases the reason for the failure is not known even after performing thorough fault injection analysis. In other cases code segments can be inherently complex to be discerned for further modifications (immunization). In such scenarios specific immunization is not possible, so we need to provide generic immunization by rebuilding the faulty component or deploying it in a new conducive environment.

As depicted in Figure 4, we download two components from remote location A and remote location B. After the download we first modify the package name so that the downloaded component can also become a part of the new component being developed. Supposing the programmer is interested in the method newnameB() after looking into the component's architecture. He simply modifies the private method to a public method and then finds out that there exists a naming collision within the same component. In order to access the variable value the method has to be made public. Now that the fault injector has made the method public with a return value, he can access that variable value by simply creating an instance of the remote object in the local component and hence being able to access that newly converted public methods' return variable value.

5.2 Evaluation results

We implemented the prototype for the component evaluation phase of our fault detection and runtime immunization approach to determine the existence of naming collisions. After we generate the source codes we perform three stages of tests to: (1) identify the variables in use; (2) ascertain the scope of each variable; and (3) determine if naming collisions will occur when their respective intermediate values are accessed.

There are two scenarios of accessing the variables in other components. Suppose component A tries to access a variable "i" in component B, and they both are in the same package, where class1 is in component A and class2 is in component B. The procedure followed to access that variable is by classname.methodname.variablename—in our example, class2.func2.i. Through this method component A will be able to access the variable "i" in component B. Still, there is a possibility that the variable "i" may not be accessible as it could be in the private member function of the component B. For this reason, we need to extend the scope of that method to public. When we extend the variable's scope there is a chance that there is another variable "i" in the same component, which is globally defined or within the same method with another initialization to the same variable. Consequently, the accessing component might be getting the last assigned value to that variable. In order to access the initial value, we will have to assign different names to those variables that cause naming collisions.

The second scenario occurs when a component is trying to access the variables from different components. Suppose component A is accessing the variable "i" from component B, as well as variable "i" from component C. The first step for the component to access the variables from different components will be to put them all into the same package. After this, we have to check the scope of the variable to determine if it is possible for another component to access this variable; if not, then we will have to extend the scope of the variable and then verify it doesn't cause any naming collisions, and then provide access to the component attempting to access that variable. Suppose class 2 is in component B and class 3 is in component C, and methods func2 is in class2 and func3 is in class3, to access the value of the "i" in component B, the code will be class2.func2.i. Similarly, the variable "i" in component C can be accessed using the code class3.func3.i. To avoid further confusion, we can assign these variables to different names after abstracting them in component A so that naming collisions do not occur in the root component.

Table 2. Naming Collision Results

Number of Components Tested	Number of Components with Naming Collision	Total Number of variables reused	Naming collisions without Scope Extension	Naming Collisions with Scope Extension	Detected and Immunized
81	37	104	30	36	66

Table 2 shows the test results for the components that were tested in our experiment. Most of the components that where tested were downloaded from IBM's Alpha works website, while the rest were from various other sources. Each component has a minimum of 100 lines of code or more.

The total number of components tested was 81. Out of the 81 components, 37 components experienced naming collision problems, both before and after their respective scopes were extended. A total of 104 variable names were reused in different scopes in the various components. Out of these 104 variables, 30 variables had scopes that were not well defined, causing naming collisions even without an extension in scope. There were a total of 36 variables that caused naming collisions after their scope was extended. We were able to detect all 66 instances where variables caused naming collisions.

6. CONCLUSION AND FUTURE WORK

Although many current systems are being developed using Java, there are also many other distributed software components developed using other technologies such as Windows COM[14] (e.g., DLLs), Unix Shared Libraries (e.g., SO files), and even .Net libraries. The .Net platform is relatively new and is a major competitor for Sun's Java. The .Net uses Intermediate Language (IL), which is very similar to the Java Intermediate Bytecode and uses a Common Language Runtime (CLR) also very similar to Java Virtual Machine (JVM) to load the code in to memory. Since .Net and Java share common object oriented model, memory models, semantics and architecture. Our instrumentation and immunization techniques can be directly applied with little modifications. In contrast, DLLs and Shared Libraries are quite different from the bytecode (intermediate code) because these are libraries in assembly code (low level). In the past there has been some research conducted in this area, and in[13] they have formulated a technique to intercept and instrument COM applications. We can apply our methodologies theoretically to these platforms but in reality we may face some technical challenges. Instrumenting Windows COM applications is more difficult than instrumenting Java bytecode because of the inherent complexity of the COM technology. Our future goal is to apply our current immunization techniques to other platforms by overcoming these complexities.

Furthermore, we consider that cyber attacks may involve tampering of existing source code to include undesired functionality, replacing or modifying a genuine component with a malicious one. Software components can be subject to two major kinds of attacks, (1) An attack involving the modification of existing source code to introduce additional malicious functionality, and, (2) An attack involving the introduction of a malicious piece of code independent of the original program that can be started when the original component is used and run independent of it (e.g. a Trojan Horse). Our goal is to detect this unauthorized integrity change in code by extending our previous work[32] and extract the malicious parts out of the component while retaining its originally expected functionality.

ACKNOWLEDGMENTS

This work was supported by the US Air Force Research Lab's extended research program, based on the SFFP (Summer Faculty Fellowship Program) award in 2004, sponsored by the Nation Research Council (NRC) and the Air Force Office of Scientific Research (AFOSR).

REFERENCE

1. Dimiter R. Avresky, Jean Arlat, Jean-Claude Laprie, Yves Crouzet. *Fault Injection for the Formal Testing of Fault Tolerance.* The Twenty-Second Annual International Symposium on Fault-Tolerant Computing, July 8-10, 1992: 345-354.
2. Abadi and L. Lamport. *Composing Specications.* ACM Transactions on Programming Languages, 15(1): 73-132, January 1993.
3. Anant Agarwal, Richard Sites and Mark Horwitz. *ATUM: A New Technique for Capturing Address Traces Using Microcode.* In Proceedings of the 13th International Symposium on Computer Architecture, 119-127, June 1986.
4. Amitabh Srivastava and Alan Eustace. "*ATOM A System for Building Customized Program Analysis Tools.*" In Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI), pages 196-205, June 1994.
5. BCEL - Bytecode Engineering Library http://bcel.sourceforge.net/

6. BIT: Bytecode Instrumenting Tool http://www.cs.colorado.edu/~hanlee/BIT/index.html
7. M. Chen, E. Kiciman, E. Brewer, and A. Fox. Pinpoint: *Problem Determination in Large, Dynamic Internet Services*. In Proceedings of the IEEE International Conference on Dependable Systems and Networks, DSN, 2002.
8. Ajay Chander, John C. Mitchell, Insik Shin. *Mobile Code Security by Java Bytecode Instrumentation*. In Proceedings of the 2001 DARPA Information Survivability Conference & Exposition (DISCEX II), pages 1027-1040, Anaheim, CA, June 2001.
9. Brian Bershad et al. *Etch Overview*. http://etch.cs.washington.edu/
10. James R. Larus and Eric Schnarr. "*EEL: Machine-Independent Executable Editing*." In proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI), pages 291-300, June 1995.
11. Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. *Techiques for efficient Inline Tracing on a Shared-Memory Multiprocessor*. In Pro-ceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modelings of Computer Systems, 8(1), May 1990.
12. A. Ghosh, J. Voas. *Inoculating Software for Survivability*. Communications of the ACM, July 1999.
13. Galen Hunt and Doug Brubacher. *Detours: Binary Interception of Win32 Functions. Proceedings of the 3rd USENIX Windows NT Symposium*, pp. 135-143. Seattle, WA, July 1999.USENIX.
14. Galen Hunt and Michael Scott. *Intercepting and Instrumenting COM Applications. Proceedings of the Fifth Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pp. 45-56. San Diego, CA, May 1999. USENIX.
15. Jikes Bytecode Toolkit - IBM Alpha Works http://www.alphaworks.ibm.com/tech/jikesbt.
16. S. Jajodia, C. McCollum, and P. Ammann. *Trusted Recovery*. Communications of the ACM, 42(7), pp. 71-75, July 1999.
17. JOIE - The Java Object Instrumentation Environment http://www.cs.duke.edu/ari/joie/
18. J. Knight, M. Elder, and X. Du. *Error Recovery in Critical Infrastructure Systems*. Proceedings of the 1998 Computer Security, Dependability, and Assurance (CSDA'98) Workshop, Williamsburg, VA, November 1998.
19. G. Kapfhammer, C. Michael, J. Haddox, R. Coyler. *An Approach to Identifying and Understanding Problematic COTS Components*. The Software Risk Management Conference, ISACC 2000.
20. J. Knight and K. Sullivan. *Towards a Definition of Survivability*. Proceedings of the 3rd Information Survivability Workshop (ISW), Boston, MA, October 2000.
21. P. Liu, P. Ammann, and S. Jajodia. *Rewring Histories: Recovering from Malicious Transactions*. Distributed and Parallel Databases, 8(1), pp. 7-40, January 2000.
22. James R. Larus and Thomas Ball. *Rewriting Executable Files to Measure Program Behavior*. Software, Practice and Experience, 24(2), February 1994.
23. H. Lipson and D. Fisher, *Survivability -- A New Technical and Business Perspective on Security*. Proceedings of the New Security Paradigms Workshop (NSPW'99), Caledon Hills, Ontario, Canada, September 21-24, 1999.
24. Henrique Madeira, Diamantino Costa, Marco Vieira. *On the Emulation of Software Faults by Software Fault Injection*. International Conference on Dependable Systems and Networks (DSN 2000). New York, New York, June 25 - 28, 2000.
25. N. Mead, R. Ellison, R. Linger, et al. *Survivability Network Analysis Method*, SEI Technical Report: CMU/SEI-00-TR-013, September 2000.
26. Amitabh Srivastava and David Wall. "*A Practical System for Intermodule Code Optimization at Link-Time.*"Journal of Programming Languages, vol 1, no 1, pages 1-18, March 1993.
27. Joon S. Park. *Component Survivability for Mission Critical Distributed Systems*. Technical Report, NRC/Air Force SFFP (Summer Faculty Fellowship Program), 2004.

28. Joon S. Park and Pratheep Chandramohan. *Component Recovery Approaches for Survivable Distributed Systems*. 37th Hawaii International Conference on Systems Sciences (HICSS-37), Big Island, Hawaii, January 5-8, 2004.
29. Joon S. Park, Pratheep Chandramohan, and Joseph Giordano. *Survivability Models and Implementations in Large Distributed Environments*. 16th IASTED (International Association of Science and Technology for Development) Conference on Parallel and Distributed Computing and Systems (PDCS), MIT, Cambridge, MA, November 8-10, 2004.
30. Joon S. Park, Pratheep Chandramohan, and Joseph Giordano. *Component-Abnormality Detection and Immunization for Survivable Systems in Large Distributed Environments*. 8th IASTED (International Association of Science and Technology for Development) Conference on Software Engineering and Application (SEA), MIT, Cambridge, MA, November 8-10, 2004.
31. Joon S. Park and Judith N. Froscher. *A Strategy for Information Survivability*. 4th Information Survivability Workshop (ISW), Vancouver, Canada, March 18-20, 2002.
32. Joon S. Park and Ravi Sandhu. *Binding Identities and Attributes Using Digitally Signed Certificates*. 16th IEEE Annual Computer Security Applications Conference (ACSAC), New Orleans, Louisiana, December 11-15, 2000.
33. Ted Romer, Geoff Voelker, Dennis Lee, Alec Wol-man, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. *Instrumentation and Optimization of Win32/Intel Executables Using Etch. In Proceedings of the 1997USENIX Windows NT Workshop*. August 1-7, 1997.
34. Jeffrey Voas. *Software Fault Injection*. IEEE Spectrum, appeared in 2000.
35. Jeffrey Voas, Keith W. Miller, and Jeffrey E. Payne. *Pisces: A tool for predicting software testability*. In the Proceedings of the Symposium on Assessment of Quality Software Development Tools, pages 297-309, New Orleans, LA, May 1992.
36. Jeffrey Voas and Jeffrey Payne. *Dependability certification of software components*. Journal of Systems and Software, 2000.