AFRL-IF-RS-TR-2007-149
Final Technical Report
June 2007

DYNAMIC HYBRID COMPONENT TEST FOR MISSION-CRITICAL DISTRIBUTED SYSTEMS

Syracuse University

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

AFRL-IF-RS-TR-2007-149 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
TANYA MACRINA Work Unit Manager	WARREN H. DEBANY, Jr. Technical Advisor, Information Grid Division Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* JUN 2007	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Jul 06 – Dec 06

4. TITLE AND SUBTITLE DYNAMIC HYBRID COMPONENT TEST FOR MISSION-CRITICAL DISTRIBUTED SYSTEMS	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER FA8750-06-1-0219
	5c. PROGRAM ELEMENT NUMBER 62702F
6. AUTHOR(S) Joon Park	**5d. PROJECT NUMBER** 4519
	5e. TASK NUMBER TM
	5f. WORK UNIT NUMBER 01
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Syracuse University 342 Hinds Hill Syracuse NY 13244-4100	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/IFGB 525 Brooks Rd Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2007-149

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 07- **297**

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The objective of this effort was to provide dynamic and hybrid survivability mechanisms that test a downloaded component in runtime in the current computing environment by considering N-category, N-type, and N-way testing methods. The test results can be used to fix the failures or immunize the component based on our previous research outcomes.

15. SUBJECT TERMS
Information systems, survivability, cyber attacks

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Tanya Macrina
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UL	28	**19b. TELEPHONE NUMBER (*Include area code)***

Table of Contents

List of Figures

List of Table

1. Introduction

The software survivability is the capability of an entity to continue its mission even in the presence of damage. An entity ranges from a single software-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. To make a system survivable it is the mission of the system to continue even in the presence of damage, rather than the individual functions of each component with full capacity all the time.

The need for survivability is most pressing for mission-critical systems. As information systems became ever more complex and the interdependence of these systems increase, the survivability picture became more and more complicated. Unfortunately, it is not always possible to anticipate every type of failure and cyber attack within large information systems, and attempting to predict and protect against every conceivable failure and attack soon becomes exceedingly cumbersome and costly. Additionally, some damage results from novel, well-orchestrated, malicious attacks that are simply beyond the abilities of most system developers to predict. Under these conditions, even correctly implemented systems do not ensure that the system is survivable. 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.

In reality, we must assume that all software components are susceptible to malicious cyber attacks or internal failures. Typically, failures are caused by poor implementation, local test criteria, different runtime environments, and so on. Cyber attacks may involve tampering with existing source code to include undesired functionality (e.g. Trojan horses), or replacing a genuine component with a malicious one. When using a downloaded component, particularly in mission-critical applications, we need to check to see if the source of the code is trusted and if the code has been modified in an unauthorized manner since it was created. Also, we need to monitor if the component is running correctly in the current computing environment. Furthermore, once we find failures or malicious codes in the component, we should fix the problems and recover the original functionality of the component so that we can support survivability in the mission-critical system.

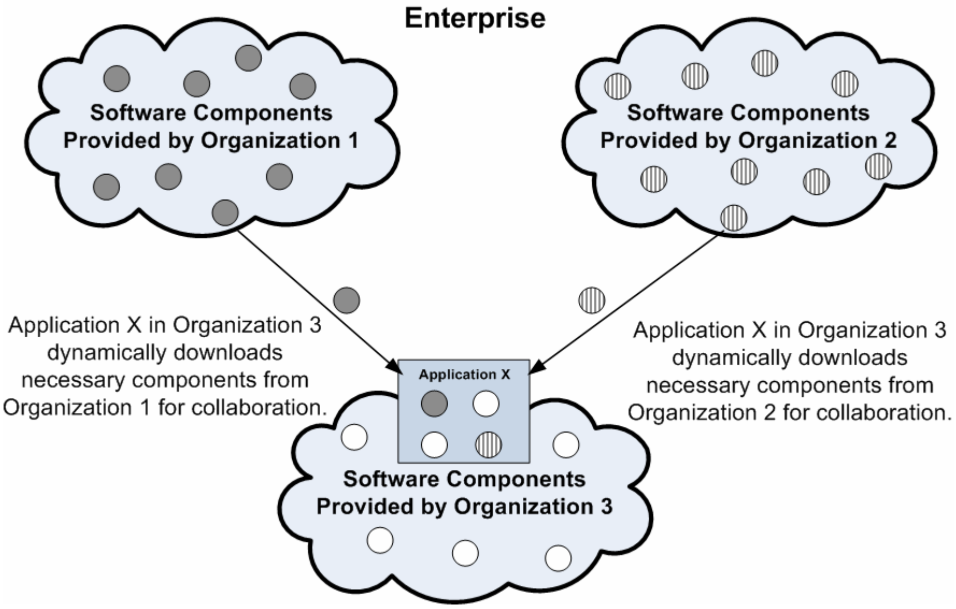
Traditional approaches for ensuring survivability, such as replay-based recovery, redundancy-based recovery, and conventional component test, do not meet the challenges of providing assured survivability in systems that must rely on commercial services and products in a distributed computing environment. Those traditional approaches can still improve the component survivability at some point, but they cannot solve the problems (i.e., internal failures or malicious codes) fundamentally, especially in runtime.
With the replay-based approach, a component may go back to the state before the failure; however, it cannot go further back into its previous state unless the reason for the failure is fixed. This means the component is not able to continue its mission in the case. Furthermore, this approach does not consider malicious codes that are already included in the components. Similarly, with the redundancy-based approach, if the redundant components are distributed in different network places, the services provided by those components can be recovered in the event of network failures, in different environments. However, if there is a failure within a component or an attack to a component, replacing that component with an identical copy is not a fundamental solution, because identical components are vulnerable to the same failure and attack. Therefore, we need advanced approaches to ensure survivability in mission-critical systems that must rely on commercial services and products in a distributed computing environment. In the following section we describe the key challenges to advanced software-survivability.

In this project we focus on the survivability of mission-critical software components downloaded on the Internet. We assume that all software components are susceptible to internal failures and malicious cyber attacks. The failures may cause because of poor implementation, local test criteria, different runtime environments, and so on. The attacks may involve tampering of existing source code to include undesired functionality (e.g. Trojan horses), replacing a genuine component with a malicious one. When using a downloaded component, particularly in mission-critical applications, we need to check if the source of the code is trusted and the code has not been modified by an unauthorized manner since it was created. Also, we need to monitor if the component is running correctly in the current computing environment. Furthermore, once we find the failures or malicious codes in the component, we should fix the problems and recover the original functionality of the component so that we can support the survivability in the mission-critical system.

In our previous projects, sponsored by the Air Force Research Laboratory, we defined our definition of survivability using an abstract state diagram, identified the static and dynamic survivability models, proposed novel approaches for component recovery and immunization in runtime based on the dynamic model, and proved the feasibility of our ideas by implementing the proposed approaches [PC04, PCDG05, PCG04a, PCG04b, PCSG07, PG07, PJG06]. In this project, as an extension of our previous work, we develop dynamic and hybrid survivability mechanisms that test a downloaded component in runtime in the current computing environment by considering multiple-aspect testing methods. The test results can be used to fix the failures or immunize the component based on our previous research outcomes.

2. Survivability Challenges in a Mission-Critical Distributed System

Typically, an application running at an enterprise level may span more than one organization. Figure 1 shows an example of a distributed application that spans multiple organizations. The figure depicts three organizations interconnected to form a large enterprise-computing environment. In the real world, there may be more than three organizations connected to form a large enterprise, and some of the organizations in the enterprise may provide specialized services that other organizations do not provide. It is each organization's responsibility to develop the corresponding software components for providing its services. In the figure, for example, Organizations1, 2, and 3 are currently involved in Application X for their enterprise work. In this example the application running in Organization 3 downloads the necessary components from Organizations 1 & 2 for some special features that it lacks. These components are dynamically downloaded in runtime from the remote hosts in Organizations 1 & 2 and run locally in Organization 3. From this point forward, the software running in Organization 3 should cooperate with the downloaded components that originated from different computing environments. By employing autonomous operation the local computing environment may have to perform some extra job of dealing with possible problems such as interoperability, failure situations, or malicious codes in the external components.


Enterprise
Software Components Provided by Organization 1
Software Components Provided by Organization 2
Application X in Organization 3 dynamically downloads necessary components from Organization 1 for collaboration.
Application X in Organization 3 dynamically downloads necessary components from Organization 2 for collaboration.
Application X
Software Components Provided by Organization 3

[Figure 1] A Distributed Application Spanning Multiple Organizations.

Typically, an application running at an enterprise level may span more than one organization. Figure 1 shows an example of a distributed application that spans multiple organizations. The figure depicts three organizations interconnected to form a large enterprise-computing environment. In the real world, there may be more than three organizations connected to form a large enterprise, and some of the organizations in the enterprise may provide specialized services that other organizations do not provide. It is each organization's responsibility to develop the corresponding software components for providing its services. In the figure, for example,

Organizations1, 2, and 3 are currently involved in Application X for their enterprise work. In this example the application running in Organization 3 downloads the necessary components from Organizations 1 & 2 for some special features that it lacks. These components are dynamically downloaded in runtime from the remote hosts in Organizations 1 & 2 and run locally in Organization 3. From this point forward, the software running in Organization 3 should cooperate with the downloaded components that originated from different computing environments. By employing autonomous operation the local computing environment may have to perform some extra job of dealing with possible problems such as interoperability, failure situations, or malicious codes in the external components.

This scenario becomes vastly more complicated when systems are integrated with Commercial off-the-Shelf (COTS) components and services, which usually have both known and unknown vulnerabilities that may cause unexpected problems that disrupt mission-critical systems. Traditional methods of ensuring survivability that deal with a set of anticipated failures and cyber attacks do not completely meet the challenges of providing assured survivability in these systems. Therefore, it is important to employ additional means to ensure proper survivability of such mission-critical systems.

Based on the typical operational scenario in a large distributed computing environment described above, we identify the following generic challenges of the component-sharing services in large distributed systems that span multiple organizations.

Challenge 1:

An autonomous mechanism to support component-sharing services in a trusted manner between different organizations or systems is needed because there is no single administrator who can control every aspect (e.g., software component development and testing) of the various systems used in an enterprise. This is an inherent challenge, especially when many systems, including those maintained by different organizations, are integrated within current distributed computing environments, which implies that a downloaded component may have failures or malicious codes that can affect the local computing environment. For instance, in Figure 1, when different organizations collaborate for a common enterprise but are still competitors in the market, they cannot simply trust the components from other organizations. Unfortunately, a remote component cannot be tested in the local environment until runtime.

Challenge 2:

Testing software components before deployment cannot detect or anticipate all of the possible failures or malicious codes that may manifest themselves during runtime, especially when external components are integrated with local components. Some failures are detected only when the components are deployed and integrated with other components in the current operational environments. Malicious codes could be added to a legitimate component after it was originally developed and tested. Existing problems in one component can be triggered by other components in runtime. Furthermore, since we cannot simply assume that all the participating organizations followed proper testing procedures of their software components, we need a new mechanism that can test the component in the actual runtime environment, especially for components downloaded from different environments. The runtime test criteria can vary, based on current applications or operational environments, even for the same component.

Challenge 3:
In a distributed mission-critical system we must check if a downloaded component has been altered in an unauthorized manner, especially if it contains malicious codes such as Trojan Horses or viruses, before malicious codes are activated in the system. For instance, in Figure 1, when different organizations collaborate for a common enterprise but are still competitors in the market, each organization should check the components from other organization before they are used in the local environment. Furthermore, if a component includes some malicious codes, but the functionality of the original component is still needed for the mission of the system, we cannot simply reject the entire component. Instead, we should retrieve only the legitimate code safely, enervating the malicious code.

Challenge 4:
According to the currently available redundancy-based approaches, we may prepare multiple copies of a critical component. However, this cannot provide survivability fundamentally. If one component has failed because of reason R1, the rest of the redundant components will fail for the same reason. It is only a matter of time until every redundant component is compromised for the same reason. Furthermore, the strength of the redundancy-based approaches depends on the prepared redundancy, which brings up the question of how many redundant components we need provide. Technically, one could maintain as many redundant components as necessary for a critical service. However, if the initially selected component is running in its normal state—meaning there is no need to use other redundant components since the component is not defective or compromised—the cost for running the redundant components has been wasted. In this situation the resource efficiency is low, and the maintenance cost is high. Therefore, a dynamic technique is needed to detect and analyze possible problems in the components and to fix those problems in runtime.

Challenge 5:
Even if we know the reasons for the software failures or the types of malicious codes, in most currently available recovery approaches in distributed computing environments, changing the components capability (e.g., component immunization) in runtime is limited, especially when the source code is not available (which is not an uncommon situation). When dealing with component failures or malicious codes in a component, one conventional way is to modify the corresponding source codes. However, this approach is possible only if the source code for that component is available. In the case of COTS components and other components downloaded from externally administered systems, the source code is often unavailable. This limitation becomes more critical when problems should be fixed in runtime in a mission-critical system. One must therefore employ advanced techniques to achieve the goal of fixing failed or compromised components in runtime—without access to the source code—in order for the mission of the component to continue.

3. Related Work

Currently, existing technologies for identifying faulty components are more or less static in nature. One of those approaches employs black-box testing of the components [AL93]. In this technique, behavioral specification is provided for the component to be tested in the target system. This technique treats the target component as a black box and can be used to determine how the component behaves anomalously. Traditionally, black-box testing is done without knowledge of the internal workings of the component tested. Normally, black-box testing involves only input and output details of the component, while information on how the output is arrived at is not needed. The main disadvantage of this technique is that the specifications should cover all the details of the visible behavior of the components, which is impractical in many situations.

Another approach employs a source-code analysis, which depends on the availability of the source code of the components. Software testability analysis [VMP92] employs a white-box testing technique that 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, which later helps him to create appropriate test data. Yet another approach is software component dependability assessment [VP00], a modification or testability analysis which thoroughly tests each component. These techniques are possible only when the source code of the components is available.

In the past, Kapfhammer et al. [KMHC00] employed a simple behavioral specification utilizing execution-based evaluation. This approach combines software fault injection [AALC96, HTI97, MCV00] at component interfaces and machine learning techniques to: (1) identify problematic COTS components, and (2) to understand these components' anomalous behavior. They isolated problematic COTS components, 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 to better understand and to increase the robustness of faulty components. In other research [VM98, CKBF02], the authors 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. In our work we employ a fault injection technique in runtime to analyze how the system behaves under injected faults.

The use of interfaces is well known across several programming languages and software programming concepts, and so are the concepts related to performance measurement, load sharing, etc. However, we are using these techniques here to build a system that is capable of taking in more testing ability at short notice and thus provide a more scalable and flexible architecture. Building tools and code modules as plug-ins that are able to incorporate additional application-specific code from the programmer at run-time have been implemented quite often in the past. We would be using this mechanism in our system to incorporate in quick time the code needed to test almost any type of threat or malicious content, apart from testing code dependencies and internal failures in components, as we shall speculate on later.

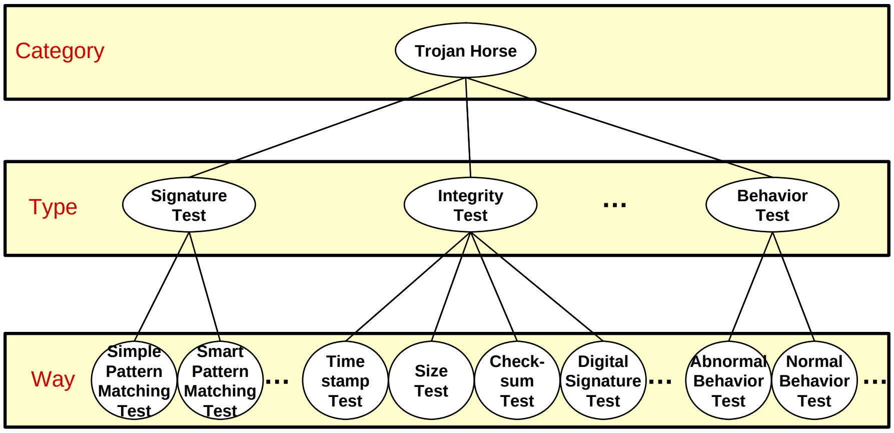
N-Version Programming (NVP) is a well-known concept in fault-tolerant systems. The NVP was proposed by [CA78, Che90, CLV05] for providing fault tolerance [AKL87, ALS88, Ran75] in software. It has been researched thoroughly during the past decade. N-Version techniques are used to ensure the reliability of a system by having multiple and different, yet functionally equivalent implementations of critical software to ensure reliability of software systems. We have suitably modified the technique here in the form of N-Way Testing, implementing it as a way of increasing the possibility of detecting a fault in a component by having as many ways as possible to conduct the test on target components. In our system we have incorporated these strategies, but in a different flavor to bring in flexibility and adequate fault tolerance, as we shall describe in more detail in the following sections.

4. Multiple-aspect Component Test

When components are downloaded from a site, one cannot immediately judge the trustworthiness of these components, especially if the provider is not in a trust ring. A significant overhead is incurred in testing for malicious or faulty contents in the downloaded components. For a mission critical system it is even more important to ensure the safety and reliability of the host system over which these safety mechanisms have been implemented. There are several issues in the way components become faulty. One possible threat is that a component may be altered or replaced with malicious contents while its transit over the network. As mentioned earlier, traditional approaches, on detecting vulnerability would simply block or prevent the faulty component from executing in the host environment. This is not the solution to our problem as mission critical systems cannot afford complete blocking of components, which were probably from a credible source but affected by malicious content during its transit (over a network, etc). According to the typical scenario of a large distributed application described in Section 2, we cannot simple assume that the source code of the downloaded component is available in the local runtime environment. Furthermore, in some mission-critical systems, the fault has to be fixed at the runtime. The entire concept of survivability hinges on the basis of providing the system with capabilities to continue execution of the component after the faulty parts have been successfully removed (or rendered harmless) by providing reliable recovery schemes. Thus, as an extension of our previous work, in this project our focus is on improving accuracy in detection of malicious contents is a downloaded component. The dynamic testing strategies mentioned below are collectively termed as *multiple-aspect testing* due to the nature of the testing itself.

In this project, we propose a multiple-aspect testing approach, which is used as a multi-functional and extendible approach to achieve more of such accuracy in detection of failure or malicious codes in a component downloaded from remote cite. The multiple aspect testing consists of the various levels of testing that the system would need to conduct, i.e., N-category testing, N-type, and N-way testing.

Figure 2 shows our multiple-aspect testing architecture for detecting Trojan horses in a component downloaded from a remote system. It may include more kinds of types and ways than those described in the figure. In our architecture we define three kinds of types and seven kinds of ways. The actual examples of these types and ways with details are described in the following sections.



[Figure 2] Multiple-Aspect Testing Architecture for Detecting Trojan Horses

N-category Testing:

A category refers to the type of faults we are addressing through our tests. For example, testing for Trojan horses would be one category of testing, while testing for internal failures would be another category. Since the system is to be implemented such that it can handle testing of a number of such categories, we call it N-category testing. Thus if the categories for a component can be defined by a set C = {c1, c2, c3… cn} then each of the elements of C can be one of virus, internal failure, Trojan horse, and so on.

N-type Testing:

The N-type refers to the type of tests that are used to detect failure or malicious codes in a download component. The N-type testing can be defined by a set of types T = {t1, t2, t3… tn} where each of the elements of T can be one of signature-based test type, integrity-based test type, behavior-based test type, and so on. For each type of test, there can be several test ways that have different but conceptually equivalent implementations. For every category defined in the set C = {c1, c2, c3…cn} we have a set of types.

The type of signature test detects Trojan horses based on a known pattern (i.e. signature). There are at least two ways in signature test type, simple pattern matching and smart pattern matching. The simple pattern matching tests the downloaded components based on a set of rules (patterns) that describe characteristics of well-known Trojan horses. Furthermore, the smart pattern matching considers more advanced patterns such as the order of the activities or the interdependencies of the components.

The type of integrity test includes four different ways such as timestamp, size, checksum, and digital (code) signature. The timestamp test checks the time intervals between requesting and receiving the component from the remote machine. If the time interval for downloading a component is greater than an acceptance threshold, the component is suspicious to be affected by malicious activities during the transit. By checking the sizes of the original and the downloaded components, we can figure out if malicious contents have been added to the original component. Since an advanced attacker can add malicious contents to the component maintaining the original size of the component, optionally we can check the checksum of the component or the code signature on the component.

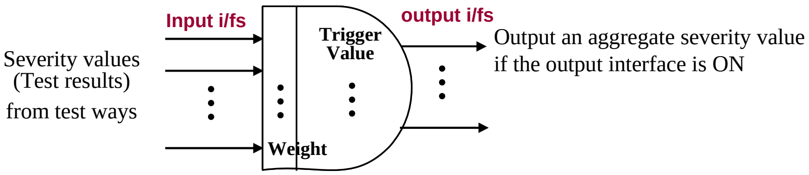
Finally, the type of behavior test builds information that describes the normal or abnormal behaviors of components to check if the component's current behavior in the local environment is expected and legitimate. The abnormal behavior test detects an abnormal behavior based on the attack state model that describes state for known attack behavior (e.g., trying to get access to boot record, buffer/heap overflow). The downloaded component is executed on a virtual machine for security test before it is included in the main program of host computer. If a behavior that occurs in the middle of the execution of the downloaded program accords with the attack state model, then the downloaded program is considered to include malicious contents. On the other hand, the normal behavior test monitors the component's current behavior based on the set of expected state model that describes the characteristics (i.e., the expected behavior when executed) of the normal components downloaded. If the downloaded component forms a poor match with the defined behaviors, then it is considered to include malicious contents.

N-way Testing:

This is in a way similar (at the same time quite different, as mentioned below) to the manner in which N-version techniques work in fault tolerant systems. If for each type of test conducted on a category, we can have several different but functionally equivalent implementations, then we have a greater chance of detecting a fault or malicious contents in the downloaded components. N-version techniques are used to ensure the reliability of a system by having multiple and different yet functionally equivalent implementations of the critical software to ensure reliability of the system. N-way testing on the other hand is a way of increasing the possibility of detecting a fault in a component by having as many ways as possible to conduct the test on the target component. Another important observation here is that unlike the N-version schemes where the decision mechanism is normally a voter when there are more than two versions and is a comparator when there are only two versions, the N-way system just triggers the recovery block even if one of the implemented "ways" detects malicious content. In this way the system is also similar to a Recovery Block mechanism commonly used in fault tolerant software systems, which executes alternate sections of code until the right result is produced.

The test mechanisms are structured in a way such that they can provide more Types and Ways of testing various Categories, depending on the intensity of the threats and the criticality of the system to be protected. Thus for a given number of Categories C, Types T, and Ways W, the total number of test modules plugged into the system would be C * T * W. Thus if the criticality of the system is high, then the number of tests to be performed on the downloaded components would have to be higher to ensure higher reliability; and would thus be a factor proportional to C * T * W. The ability to handle higher criticality of the host machine can also be implemented in the system simply by increasing the acceptance threshold and the critical acceptance threshold defined earlier.

Based on the multiple-aspect testing, we propose a cooperative testing scheme, in which the test ways cooperate with each other to provide faster detection capability. Before explaining the cooperative scheme, we introduce a Trigger Decision Gate (TDG), which has been developed in order to express the relation among test ways.

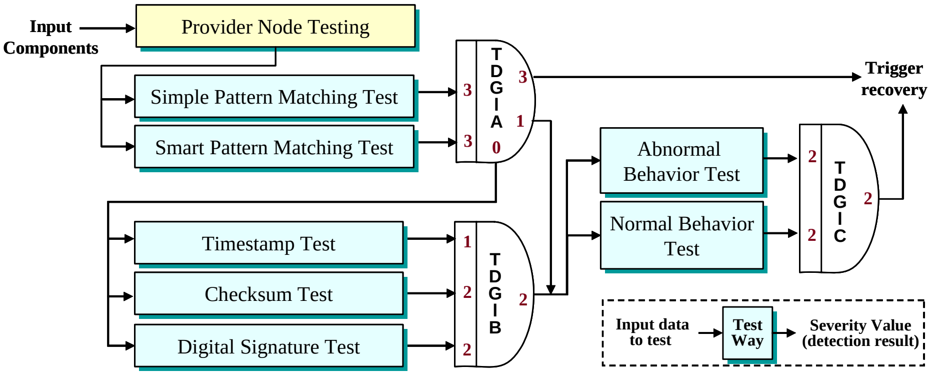

Input i/fs
output i/fs
Trigger Value
Output an aggregate severity value if the output interface is ON
Severity values (Test results) from test ways
Weight

[Figure 3] Notation of Trigger Decision Gate (TDG)

Figure 3 shows the notation of TDG that collects test results (i.e., severity values) through input interfaces from test ways, and then outputs an aggregate severity value to the corresponding output interface. In Figure 3, weight means a priority value assigned to the corresponding input interface. Trigger value in TDG indicates a minimum value for turning the corresponding output interface on. The output interface is triggered as follows:

$$\text{OutputInterface}_{\text{k}} = \begin{cases} ON, \text{ if } \left(TriggerValue_{\text{k}} \le \sum_{\text{i=1}}^{\text{input i/f \#}} SeverityValue_{\text{i}} < TriggerValue_{\text{k+1}} \right) \\ \qquad \text{where, } TriggerValue_{\text{k}} < TriggerValue_{\text{k+1}} \\ Off, \text{ otherwise} \end{cases}$$

Cooperative test scheme is a heuristic-based technique because we design a cooperative architecture through the analysis of the characteristics (i.e., merit and demerit) of each test type and way. For example, the signature test type has generally a merit that is able to detect malicious codes fast, but a demerit that cannot detect unknown malicious codes. On the other hand, the behavior test type can detect unknown malicious codes, but is very slow in the detection speed. The integrity test type is able to detect malicious codes done by an illegal attacker more accurately than any other test ways, but has difficulty in detecting not only malicious codes (e.g., backdoor) intentionally written by a legitimate but malicious user, but also failure codes accidentally written by a legitimate user.



[Figure 4] Cooperative Test Architecture

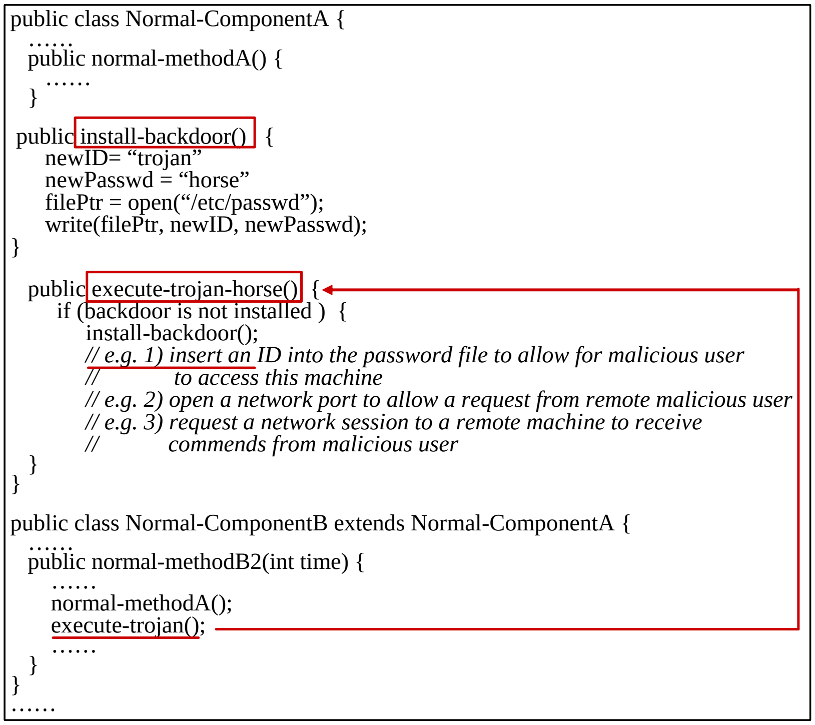
Fig. 4 shows a cooperative test architecture that is designed based on the existing schemes (i.e., pattern matching-based detection scheme, integrity-based detection scheme, and anomaly detection scheme) and the two schemes (i.e., provider node testing scheme and multiple-aspect testing scheme) proposed in this paper, which is able to enhance test performance in terms of detection accuracy and detection speed. The cooperative architecture operates as follows. When a consumer node receives several components, it first of all performs the provider node testing to choose the cleanest component among those components. And then, the consumer tests the component by using the simple pattern matching test way and smart pattern matching test. If either of the two tests detects malicious code (i.e., if aggregate severity value is greater than or equal to 3, in TGD-A), then a recovery mechanism is triggered to recover the infected component. In this paper, we do not address the recovery mechanism. Else if either of the two test ways find something suspected as malicious code (i.e., if aggregate severity value is greater than or equal to 1, in TGD-A) in the component, then it triggers performing an abnormal behavior test way and a normal behavior test. Otherwise, it triggers executing three kinds of test ways, timestamp test, checksum test, and digital signature test. If the aggregate severity value outputted from the three tests is greater or equal to 2, then TDG-B performs the abnormal behavior test way and the normal behavior test. Finally, if the aggregate severity value outputted from those two test ways is greater or equal to 2, then it triggers the recovery mechanism.

5. Applying the Proposed Approaches to Real Systems

We have implemented a prototype based on the proposed multiple-aspect testing that is capable of detecting Trojan horses in a component download from a remote system. In this section, we describe the architecture and several algorithms for the multiple-aspect testing.

5.1. Malicious Component Code

The remote component downloaded from a remote system may have failures or malicious codes such as a computer virus, worm, or Trojan horse. In this project, we address only Trojan horse as an example. The multiple-aspect testing for detecting the other malicious attacks is now under implementation based on the framework we described in the previous sections. Based on the typical scenario that we described in Section 2, we consider that a downloaded component contains a Trojan horse, especially when the participating organizations are competitors in the market.



```
public class Normal-ComponentA {
  ……
  public normal-methodA() {
    ……
  }

 public install-backdoor()  {
    newID= “trojan”
    newPasswd = “horse”
    filePtr = open(“/etc/passwd”);
    write(filePtr, newID, newPasswd);
}

  public execute-trojan-horse()  {
      if (backdoor is not installed )  {
          install-backdoor();
          // e.g. 1) insert an ID into the password file to allow for malicious user
          //            to access this machine
          // e.g. 2) open a network port to allow a request from remote malicious user
          // e.g. 3) request a network session to a remote machine to receive
          //           commends from malicious user
  }
}

public class Normal-ComponentB extends Normal-ComponentA {
  ……
  public normal-methodB2(int time) {
     ……
     normal-methodA();
     execute-trojan();
     ……
  }
}
……
```

[Figure 5] An Example of Malicious Component with a Trojan Horse

Figure 5 shows a Java-like program of a component, which includes a Trojan horse, execute-trojan-horse() method. The actual name of the malicious method can be different and its mission varies. For instance, if the execute-trojan-horse() method is called, it installs a backdoor to allow a malicious user who plants the Trojan horse to get access to the system. There may be various types of backdoors. For example, the Trojan horse code may insert a login ID and password into the password file or open a network port to allow a request from remote malicious user. As another example, it may request a network session directly to a remote machine to receive commends from malicious user. In the example of Figure 5, Trojan horse attacker uses a backdoor that inserts illegal identifier into the password file (e.g., “/etc/passwd” in Unix). Technically, an attacker can use a separate component as a Trojan horse, but in this example we consider an embedded Trojan horse in a legitimate component because we believe this case is more difficulty to detect.

If we could have access to the source code of the downloaded component it may be easier to detect the Trojan horse. However, as we discussed in Section 2, the source code is often unavailable to the local environment. Moreover, even though the source code is available, when we consider a mission-critical system we may not have sufficient time to fix the source code, recompile it, and redeploy the component in the runtime. Therefore, we need an advance approach for component test with a high accuracy that can be done on the fly without the source code.

5.2. Overall Implementation Summary

All the test algorithms used in our implementation utilize .Net Reflection techniques and the XML file mentioned earlier to validate the integrity and authenticity of the components in the collaboration system. The following figure shows the general structure of the test algorithms. Here we use a scanner module, which uses reflection (when required) to scan the component for all its types and compares this information with the XML that is provided by the publisher of the component. Obtaining the size and timestamp of a component does not require the use of Reflection and so is used selectively. Thus although algorithms have their own implementations of what they validate (size, timestamps, behavior, etc), the use of reflection to scan the component while validating the information obtained against that provided by the publisher is the common approach. We have implemented the test ways that we described with Figure 2 to test whether the downloaded component includes a Trojan horse.

In most cases we use Reflection to scan the components but as already mentioned a scan for the size, timestamps etc can be done without using Reflection. The algorithms mainly validate this information with that provided by the publisher in the XML file. The system returns a true (pass) or false (fail) depending on whether the validation is a success or not. The Reflection technique is not just used in the testing algorithms as mentioned above. One of the prime uses of Reflection is to help in the loading of test procedures at run-time. The testing algorithms are built and stored as DLLs and their functionality is loaded and executed at run-time by gathering and making calls on their types (classes and functions, etc). This process of dynamic loading and executing is achieved using Reflection. Addition or removal of the DLLs (from where it is scanned for the dynamic loads) results in those tests being performed (or not performed). Note that this separates the system from the tests that it conducts on components. This prevents breaking of the code in the system with addition or removal of test procedures; and also gives the flexibility to control the system at run-time. As a result this system is also automatically scalable to incorporate many ways at run-time (as many as the test server running we can handle without too much loss in speed and performance).

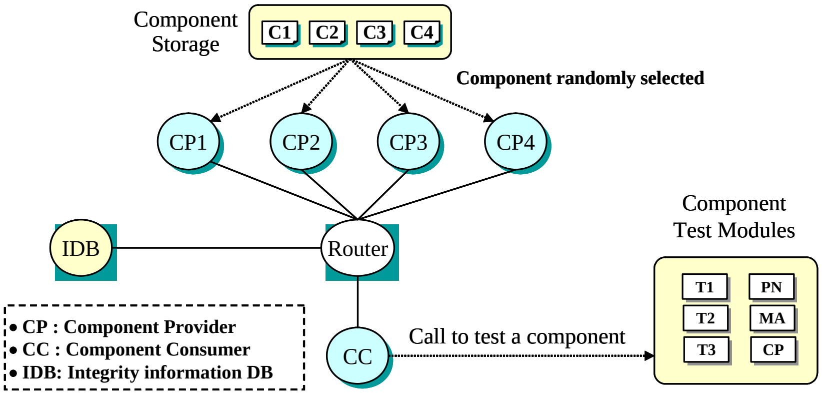
For the purpose of demonstrating the viability of using an extendible system, we developed a prototype for the same, which lays out the basic design and structure for the system. It is to the methods define only a demonstrative functionality for testing any of the categories (in our case Trojans), and is meant to demonstrate the flexibility and robustness gained in testing for survivability in components by using this model. We have considered the use of Microsoft's .Net and C# as the platform to describe and prototype the testing and loading mechanisms, as it is a new programming model that is quickly growing in popularity.

6. Performance Evaluation

In this section, we analyzes and evaluate the performance of the provider node testing scheme, the multiple-aspect testing, and the cooperative testing in terms of detection accuracy and speed. For this, we have implemented a component-sharing environment by using NS (Network Simulator) [15].

6.1 Simulation Environment

Fig. 6 shows a simulated network architecture for component testing. The architecture consists of 5 parts, component storage, Component Provider (CP) nodes, and a Component Consumer (CC) node, component test modules, and Integrity information DB.


Component Storage
C1
C2
C3
C4
Component randomly selected
CP1
CP2
CP3
CP4
IDB
Router
Component Test Modules
T1
PN
T2
MA
T3
CP
• CP : Component Provider
• CC : Component Consumer
• IDB: Integrity information DB
CC
Call to test a component

[Figure 6] Simulated network architecture for component testing

There are four kinds of components in the component storage, a normal component (C1), a component with well-known Trojan horse (C2), a component with mutative Trojan horse (C3), and component with unknown Trojan horse (C4). Note that all the components are the same each other in that they provide the identical function, but different from each other in that C1 is a normal component while C2, C3, and C4 each is infected with different type of Trojan horse. CP nodes (i.e., CP1, CP2, CP3, and CP4) each randomly selects one among the four types of components and send it to the CC node whenever it receives a request from the CC node. If the CC node downloads components from CP nodes, it tests them. We have implemented the existing three kinds of test ways, pattern-matching-based test, checksum-based test, and abnormal behavior-based test, and the three schemes proposed in this paper, provider node testing, multiple-aspect testing, and cooperative testing. The integrity information DB is used by checksum-based test way to get integrity information for a download component.

Fig. 7 shows an example of a malicious component stored in component storage of Fig. 6. The component is written in Tcl language. The code of the component provides a sin function (i.e., *remote-sin* in Fig. 7), but includes a backdoor (i.e., in *calculate-sin*). Whenever the *remote-sin* function is called, it calls the *calculate-sin* function in order to calculate the sin value for the input value. Once the *calculate-sin* function is called, it calculates the sin value for the input value, and then checks a variable, *backdoor_* if its value is 1. If the value of *backdoor_* is not 1, then the *calculate-sin* function connects to a malicious server to download a backdoor program and install it on its system.

```
set component_(3) {
    proc remote-sin {k} {
         calculate-sin $k res
        # puts "<Component> sin($k) is $res"
         return $res
    }
    global backdoor_
    proc calculate-sin {invalue outvalue} {
          upvar $outvalue result_
          global backdoor_
          set result_ [expr sin($invalue)]
           # Backdoor
          if { ![info exists backdoor_] } {
               set backdoor_ 1
               # connects to a malicious server to download a backdoor to install on the victim
               set s [socket-client maliciousHostIP 2540]
               puts $s "<Tester System> Send me the backdoor software"
               set program [gets $s]
               # install the program
               close $s
}    }    }
```

[Figure 7] Example of a malicious component stored in component storage. This is written in Tcl language and includes a backdoor in the *calculate-sin* function.

We have introduced the existing three kinks of test ways, PM (Pattern-Matching)-based way, CS (Check-Sum)-based way, and AB (Abnormal Behavior)-based way. The PM-based way uses attack signature to detect malicious code in the component shown in Fig.7. For example, if the PM-based way has "*backdoor_*" as attack signature, it will succeed in detecting the backdoor because the component defines *backdoor_* as a variable for the backdoor. The AB-based way detects malicious code by monitoring abnormal behavior during the execution of the component. So, if the AB-based way regards what the component connects to an external system as abnormal behavior, it will detect the backdoor. Finally, the CS-based way detects malicious code by comparing the checksum calculated from the component with the checksum downloaded from IDB, irrespective of the content of the component. If the value of both checksums is different each other, the CS-based way regards the component as malicious component. In this experiment, the PM-based way has not "*backdoor_*" as an attack signature, but the AB-based way defines what a component connects to an external system as abnormal behavior. So, AB-based way is able to detect the backdoor in the component shown in Fig. 7.

6.2 Analysis of Simulation Results

In this simulation, there are three kinds of users, attacker, malicious user, and normal user. An attacker is one who makes a malicious component illegally without any permission. On the other hand, a malicious means one who is a legitimate user but inserts malicious codes (e.g., Trojan horse or backdoor) into a normal component by accident or intentionally.

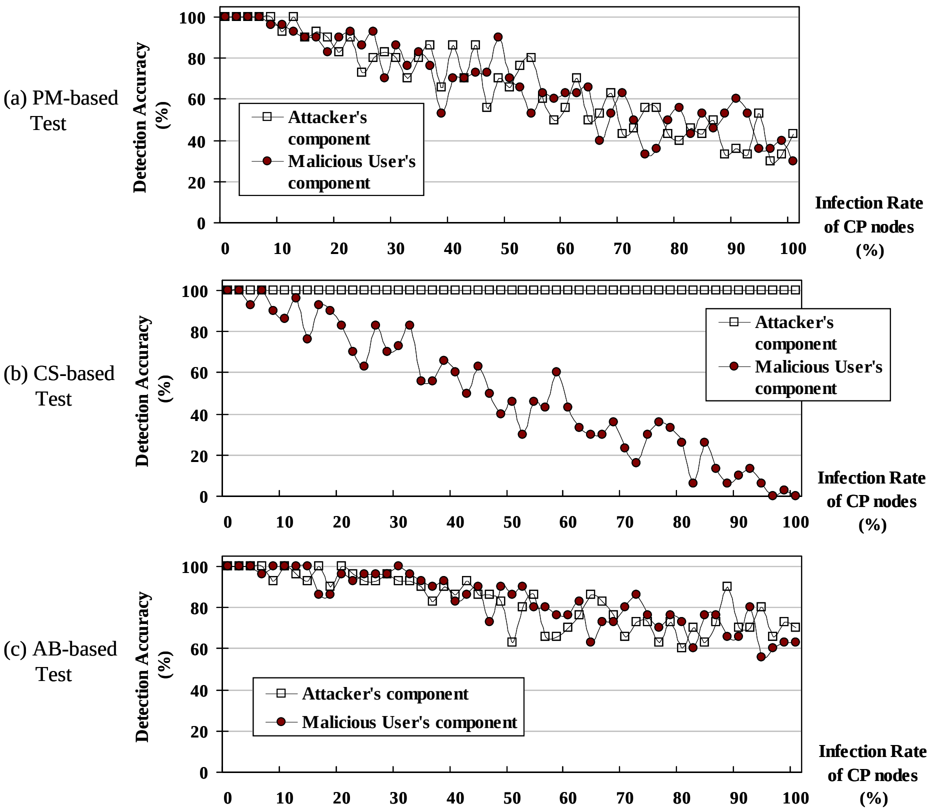
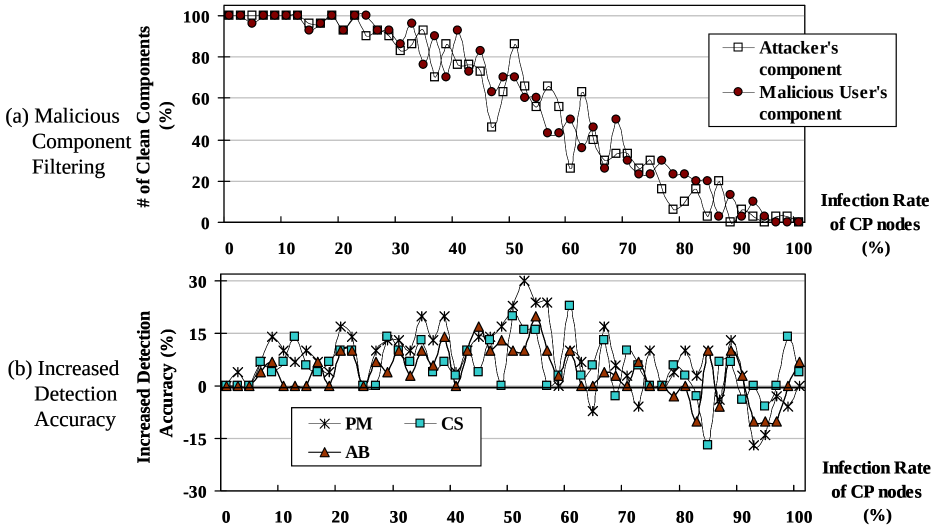


Fig. 8. Performance of Trojan horse detection in the existing component testing schemes: In this graph, attacker is one who makes a component illegally without any permission.

On the other hand, a malicious user means one who is a legitimate user but makes a malicious component. The infection rate of the CP (Component Provider) nodes indicates the rate of CP nodes infected by attackers and malicious users.

Fig. 8 shows the performance of Trojan horse detection in the existing component testing schemes. Fig. 8-(a), (b), and (c) are the performance of PM (Pattern-Matching) way, CS (Check-Sum), and AB (Abnormal Behavior), respectively. The PM way and AB way decrease in detection accuracy in proportion to the infection rate of the component provider nodes as shown in Fig. 8. On the other hand, the CS way has great advantage in detecting malicious components made by attackers, but is very poor at detecting malicious components made by malicious users. This is because attackers have no right to generate checksum for components that they create illegally, whereas malicious users are legitimate user who can generate it.

The performance of each test scheme shown in Fig. 8 has meaning in only itself and no concern with that of the other schemes. So, Fig. 8 does not mean that the AB way is better than the PM way in detection ability.

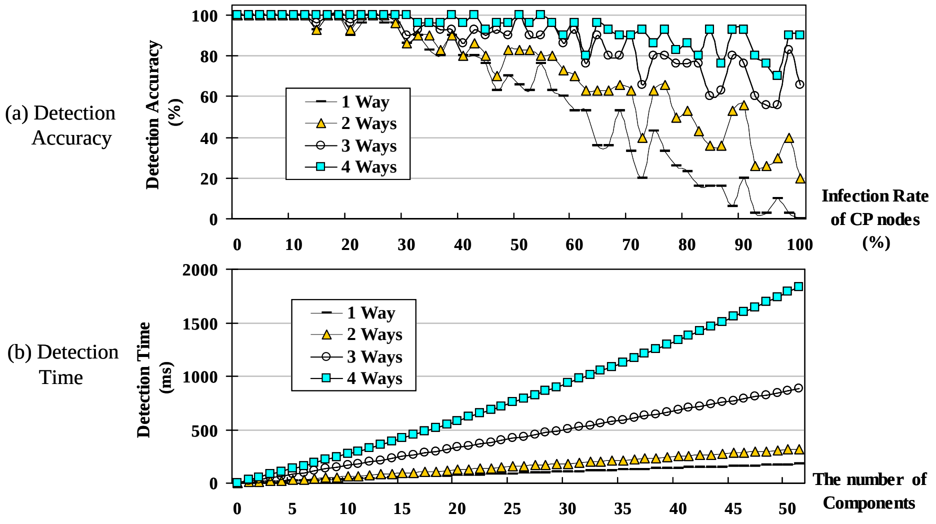

(a) Malicious Component Filtering

(b) Increased Detection Accuracy

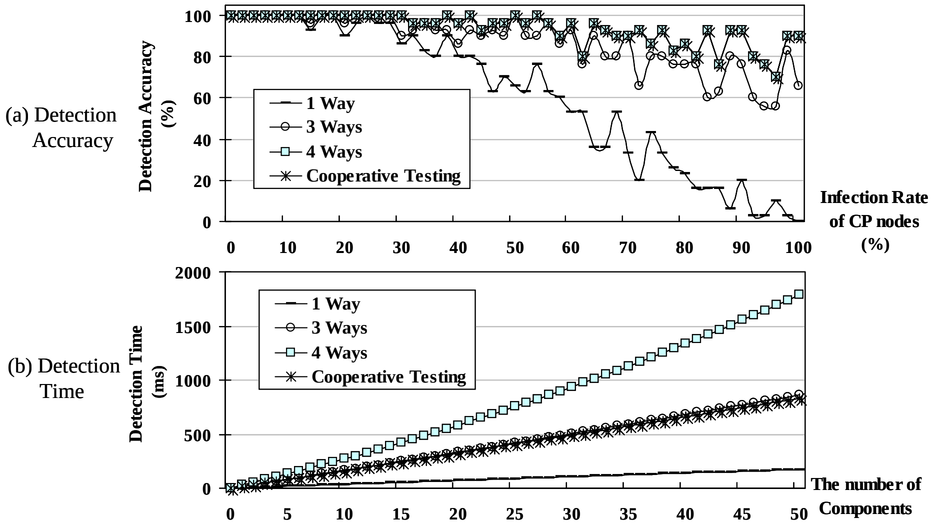
[Figure 9] Performance of Trojan horse detection in Provider Node testing scheme

Fig. 9 shows the performance of Trojan horse detection in the provider node testing scheme proposed in this paper. The provider node testing is used to increase the possibility to choose the most clean (uninfected) component among components that exist on CP nodes by filtering malicious components as shown in Fig. 9-(a). Fig. 9-(b) shows the performance of Trojan horse detection when the existing schemes employ the provider node testing scheme. The provider node testing scheme improves the detection accuracy of all the existing schemes as shown in Fig. 9-(b). The provider node testing scheme is very effective until the infection rate of the CP nodes is less then 60%, as shown in Fig. 9-(b).

Fig. 10 shows the performance of Trojan horse detection in multiple-aspect testing scheme. As shown in Fig. 10-(a), the multiple-aspect testing gains significantly in accuracy of testing over one-way techniques used for detecting malicious content. The detection accuracy in multiple-aspect testing is directly proportional to the number of test ways. This shows that the multiple-aspect testing proposed in this paper provides a dramatically high precision of the malicious code detection in a downloaded component. However, the multiple-aspect testing has a performance overhead problem because it performs several test ways to test one component. As shown in Fig. 10-(b), the detection time of the multiple-aspect testing is not good.



[Figure 10] Performance of Trojan horse detection in Multiple-Aspect testing scheme



[Figure 11] Performance of Trojan horse detection in Cooperative testing scheme

We have proposed the cooperative testing scheme to address the detection speed problem of the multiple-aspect testing. The cooperative testing scheme can provide faster detection capability by making the test ways cooperate with each other without giving much impact on its system. Fig. 11 shows that the cooperative testing scheme can detect attacks as accurately as 4-ways as shown in Fig. 11-(a), while its attack detection time is even less then 4 ways.

Table 1. Comparison between schemes proposed in this paper

Schemes Performance	Provider Node Test	Multiple-aspect Test	Cooperative Test

Attack Detection Time	-	Increase (Bad)	Decrease (Good)
Attack Detection Precision	Increase (Good)	Increase (Good)	Increase (Good)

Table 1 shows a comparison based on the simulation results between schemes proposed in this paper.

7. Conclusions and Future Work

In our previous projects, sponsored by the Air Force Research Laboratory, we defined our definition of survivability using an abstract state diagram, identified the static and dynamic survivability models, proposed novel approaches for component recovery and immunization in runtime based on the dynamic model, and proved the feasibility of our ideas by implementing the proposed approaches [PC04, PCDG05, PCG04a, PCG04b, PJG06]. In this project, as an extension of our previous work, we develop dynamic and hybrid survivability mechanisms that test a downloaded component in runtime in the current computing environment by considering multiple testing methods. The test results can be used to fix the failures or immunize the component based on our previous research outcomes.

Our simulation results proved that the proposed ideas are able to effectively and efficiently detect malicious codes in a downloaded component. We also have implemented a prototype collaboration network environment to evaluate the performance of those schemes proposed in this project in terms of detection accuracy and detection speed. The experimental results show that the provider node testing can increase the possibility to choose the most clean (uninfected) component among components that exist on multiple remote systems, the multiple-aspect testing can improve ability to detect a fault or malicious contents, and the cooperative testing scheme can provide fast detection speed. Currently, we have been implementing our approaches on real systems in Microsoft's .Net environment. Our future work will extend our idea so as to integrate for detecting various kinds of attacks as well as internal errors.

8. PI’s Publications in the Topic Area

Based on the outcomes from the PI’s research in the area of component survivability, sponsored by the US Air Force Laboratory, Rome, NY, we have published the following papers in the research journals, book chapter, conferences, and workshops.

- Gaeil An and Joon S. Park. Cooperative component testing architecture in collaborating network environment. *Proceedings of the 4th International Conference on Autonomic and Trusted Computing (ATC), Lecture Notes in Computer Science (LNCS),* Hong Kong, China, July 11-13, 2007. Springer.

- 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.

- Joon S. Park, Pratheep Chandramohan, and Joseph Giordano. *Survivability Models and Implementations in Large Distributed Environments*. Accepted for the 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.

- Joon S. Park, Pratheep Chandramohan, and Joseph Giordano. *Component-Abnormality Detection and Immunization for Survivable Systems in Large Distributed Environments.* Accepted for the 8th IASTED (International Association of Science and Technology for Development) conference on Software Engineering and Application (SEA), MIT, Cambridge, MA, November 8-10, 2004.

- Joon S. Park, Pratheep Chandramohan, Ganesh Devara jan, and Joseph Giordano. Trusted component sharing by runtime test and immunization for survivable distributed systems. In the *20th IFIP International Conference on Information Security (IFIP/SEC 2005)*, Chiba, Japan, May 30 - June 1, 2005.

- Joon S. Park, Pratheep Chandramohan, Avinash T. Suresh, and Joseph Giordano. Component survivability for mission-critical distributed systems. *Journal of Automatic and Trusted Computing (JoATC)*, 2007. In press.

- Joon S. Park, Pratheep Chandramohan, Artur Zak, and Joseph Giordano. *Fine-Grained, Scalable, and Secure Key Management Scheme for Trusted Military Message Systems.* Accepted for the Military Communications Conference (MILCOM), Monterey, CA, October 31-November 3, 2004.

- Joon S. Park and Joseph Giordano. Software component survivability in information warfare. In *Encyclopedia of Information Warfare and Cyber Terrorism*. IDEA Group Publishing, 2007. In press.

- Joon S. Park and Judith N. Froscher. *A Strategy for Information Survivability*. 4th Information Survivability Workshop (ISW), Vancouver, Canada, March 18-20, 2002.

- Joon S. Park, Gautam Jayaprakash, and Joseph Giordano. Component integrity check and recovery against malicious codes. In *IEEE International Workshop on Trusted and Automatic Computing Systems (TACS)*, Vienna, Austria, April 18-20, 2006.

References

[AALC96] D. R. Avresky, J. Arlat, J.-C. Laprie, and Y. Crouzet. *Fault injection for formal testing of fault tolerance*, IEEE Transactions on Reliability, vol. 45, no. 3, pp. 443–455, 1996.

[AL93] M. Abadi and L. Lamport. *Composing specifications*, ACM Transactions on Programming Languages and Systems, vol. 15, no. 1, pp. 73–132, 1993.

[AKL87] A. Avizienis, H. Kopetz, and J. C. Laprie, editors. *The Evolution of Fault-Tolerant Computing*. Springer, Wien, New York, 1987.

[ALS88] A. Avizienis, M. R. Lyu, and W. Schuetz. *In search of effective diversity: a six-languagestudy of fault-tolerant flight control software*. In Digest of 18th FTCS, pages 15–22, Tokyo, Japan, June 1988.

[AP07] Gaeil An and Joon S. Park. Cooperative component testing architecture in collaborating network environment. *Proceedings of the 4th International Conference on Autonomic and Trusted Computing (ATC)*, *Lecture Notes in Computer Science (LNCS)*, Hong Kong, China, July 11-13, 2007. Springer.

[CA78] L. Chen and A. Avizienis. *N-version programming: a fault-tolerance approach to reliability of software operation*. In Digest of 8th FTCS, pages 3–9, Toulouse, France, June 1978.

[Che90] J. J. Chen. *Software Diversity and Its Implications in the N-Version Software Life Cycle*. PhD dissertation, UCLA, Computer Science Department, 1990.

[CKBF02] 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.

[CLV05] X. Cai, M. R. Lyu, and M. A. Vouk, *An experimental evaluation on reliability features of n-version programming.* The 16th IEEE International Symposium on Software Reliability Engineering. Washington, DC, USA: IEEE Computer Society, November 2005, pp. 161–170.

[HTI97] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. *Fault injection techniques and tools*, Computer, vol. 30, no. 4, pp. 75–82, 1997.

[KMHC00] G. Kapfhammer, C. Michael, J. Haddox, and R. Colyer. *An approach to identifying and understanding problematic COTS components*, in The Software Risk Management Conference (ISACC), Reston, Virginia, 2000.

[MCV00] H. Madeira, D. Costa, and M. Vieira. *On the emulation of software faults by software fault injection*, in International Conference on Dependable Systems and Networks (DNS). Washington, DC, USA: IEEE Computer Society, 2000, pp. 417–426.

[PC04] 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.

[PCG04a] Joon S. Park, Pratheep Chandramohan, and Joseph Giordano. *Survivability Models and Implementations in Large Distributed Environments*. Accepted for the 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.

[PCG04b] Joon S. Park, Pratheep Chandramohan, and Joseph Giordano. *Component-Abnormality Detection and Immunization for Survivable Systems in Large Distributed Environments.* Accepted for the 8th IASTED (International Association of Science and Technology for Development) conference on Software Engineering and Application (SEA), MIT, Cambridge, MA, November 8-10, 2004.

[PCDG05] Joon S. Park, Pratheep Chandramohan, Ganesh Devara jan, and Joseph Giordano. Trusted component sharing by runtime test and immunization for survivable distributed systems. In the *20th IFIP International Conference on Information Security (IFIP/SEC 2005)*, Chiba, Japan, May 30 - June 1, 2005.

[PCSG07] Joon S. Park, Pratheep Chandramohan, Avinash T. Suresh, and Joseph Giordano. Component survivability for mission-critical distributed systems. *Journal of Automatic and Trusted Computing (JoATC)*, 2007. In press.

[PCZG04] Joon S. Park, Pratheep Chandramohan, Artur Zak, and Joseph Giordano. *Fine-Grained, Scalable, and Secure Key Management Scheme for Trusted Military Message Systems.* Accepted for the Military Communications Conference (MILCOM), Monterey, CA, October 31-November 3, 2004.

[PG07] Joon S. Park and Joseph Giordano. Software component survivability in information warfare. In *Encyclopedia of Information Warfare and Cyber Terrorism*. IDEA Group Publishing, 2007. In press.

[PF02] Joon S. Park and Judith N. Froscher. *A Strategy for Information Survivability*. 4th Information Survivability Workshop (ISW), Vancouver, Canada, March 18-20, 2002.

[PJG06] Joon S. Park, Gautam Jayaprakash, and Joseph Giordano. Component integrity check and recovery against malicious codes. In *IEEE International Workshop on Trusted and Automatic Computing Systems (TACS)*, Vienna, Austria, April 18-20, 2006.

[Ran75] B. Randell. *System structure for software fault-tolerance*. In IEEE Transactions on Software Engineering, SE-1(6):220–232, June 1975.

[VMP92] J. M. Voas, K. W. Miller, and J. Payne. *PISCES: A tool for predicting software testability*, NASA, Tech. Rep., 1992.

[VM98] J. M. Voas and G. McGraw. *Software Fault Injection: Innoculating Programs Against Errors*. Wiley Computer Publishing, 1998.

[VP00] J. M. Voas and J. Payne, *Dependability certification of software components*, Journal of Systems and Software, vol. 52, no. 2-3, pp.165–172, 2000.

Acronym List

Checksum (CS)
Abnormal Behavior (AB)
Commercial Off-the-Shelf (COTS)
Component Consumer (CC)
Component Provider (CP)
Dynamic-Link Library (DLL)
Network Simulator (NS)
N-Version Programming (NVP)
Pattern Matching (PM)
Trigger Decision Gate (TDG)