APPLICATION SECURITY AUTOMATION

Approved by:

Dr. Sukumaran V.S. Nair

Dr. Frank Coyle

Dr. Mitchell A. Thornton

Dr. Jeff Tian

Dr. Dinesh Rajan

APPLICATION SECURITY AUTOMATION

A Praxis Presented to the Graduate Faculty of the

School of Engineering and Applied Science

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Engineering

with a

Major in Software Engineering

by

Majid A. Malaika

(B.S.C.S, King Abdul-Aziz University, 2001)
(M.S.C.E, Southern Methodist University, 2007)

Dec 17, 2011

ACKNOWLEDGMENTS

Malaika , Majid A. B.S.C.S, King Abdul-Aziz University, 2001
M.S.C.E, Southern Methodist University, 2007

Application Security Automation

–

Advisor: Professor Sukumaran V.S. Nair

Doctor of Engineering degree conferred Dec 17, 2011

Praxis completed Dec 17, 2011

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

This praxis is dedicated to my wife and son...

Chapter 1

INTRODUCTION

With todays high demand for online applications and services running on the Internet, Intranet, and the Cloud software has become a vital piece in our business and personal lives. Software has become a critical component to our health, finances, political and social interaction, which is magnificent in facilitating our lives, but with every revolutionary technology comes challenges unique to its characteristics; for online applications, Security is a huge concern and challenge.

Nowadays, online applications are in serious need for better security schemes than ever without degrading performance or eliminating desired functionalities. Despite the traditional testing and quality assurance performed on applications and services today during the development phases. Vulnerabilities persist undetected. As a result, producing services and systems with severe vulnerabilities that can be exploited by cyber-criminals.

In the past few years, we have seen rapid growth and expansion in cloud computing power, mobility, and connectivity. Non-functional requirements have become more challenging and confusing. With the advanced developing tools (most of which are free), and cheap hardware and services, it is becoming easier than ever to develop software and applications by any person with minimal expertise. One booming example is the Applications (Apps) market in smartphones. Such as the Apple and Android App stores. This availability of tools, quick distribution, and high integration introduces many issues in software quality. Security is the most affected among all [2, 20, 6].

On the other hand, online services, applications and integration solutions have opened endless opportunities for enterprises, small businesses, and individuals by integrating complex, disparate systems in a very intricate fashion. In addition, new systems have been developed with an eye on the future to equip them with innovative integration capabilities. In this praxis, we investigate the problem of application security by introducing the N-Version Application Security Automation (NVASA) Framework to alleviate programming language and compiler errors produced during the Software Development Life Cycle (SDLC).

1.1. The Security Problem

During the Software Development Life Cycle, most of the non-functional requirement's deliberation is focused on ensuring the performance, maintainability, reliability, availability, and reusability of the application. However, security is usually addressed later during the SDLC, and in most cases later after production [93]. This delay increases the complexity of solving the security issues; which as a result, increases the cost and effort spent to actually fix these security issues.

Today, one of the most challenging issues facing system security is that it is viewed by enterprises as a commodity, where security is understood as an add-on feature that can be added at any stage of the SDLC or after production. Therefore, the usage of password patterns and the integration of anti-virus applications, anti-phishing tools, and firewalls promote a false sense of security if the applications running on the network werent designed with security in mind. For example, a company can install an advanced firewall, one of the best in the market, but permit users (employees) to connect to an application that contains a remotely exploitable vulnerability through a specific port on the firewall. In this case, the firewall would not protect the company from cyber-attackers gaining unauthorized access through the exploitable application;

therefore, the attackers would gain access to the hosting machine which would grant them unauthorized access to the entire network behind the firewall [107].

Most annual security reports demonstrate insufficient application security measures taken by both enterprises and individuals [52, 75, 76]. Lately, Apple released 130 security updates to fix vulnerabilities discovered in their Mac OS X 10.6.5 [5, 98, 54]. Some security experts claim that Apple still failed to fix all severe vulnerabilities and that there are few un-patched vulnerabilities in their Mac OS X 10.6.5.

Today, Information Technology (IT) security is challenged in many new ways. When the security of a company/enterprise fails, it negatively involves everyone in the business cycle from consumers, providers down to employees, and partners. Some of the factors contributing into challenging security [107, 94, 96] include:

1. **System complexity:** online applications have grown exponentially in complexity in the past few years; meanwhile, the software development life cycle has become shorter due to the rapid growth of the Internet. Applications today have the capability to interact automatically with users and/or other systems in a very complex fashion. Therefore, increasing the possibility of error injection during the planning, design, development, deployment, maintenance, or upgrade phases.

2. **Ubiquitous networking:** with the growth of the Internet connections and speeds, more systems are connected to the Internet without the appropriate security, thus becoming available online. This, in turn, increases the exposure of these systems which increases the likely hood of being exploited. Consequently, the risk associated with these unprotected applications increase.

3. **Built-in extensibility:** this is a desired feature in software engineering because it would enable the flexibility to add new components to the existing system in the future. On the contrary, this is a challenging feature in application

security because it complicates the analysis and testing of application security. In addition, it is very difficult to prevent malicious code from being injected into the system in the future.

4. **Common platforms (Software Monoculture):** while common platforms reduce cost and time when implementing new technologies or building an application, they also increase a malicious users chances of exploiting more systems and having a larger impact when they succeed. In a recent OS Web Statistics and Trends report shown in Table 1.1 [109] found that almost 90% of connected machines on the Internet were running Windows operating system. The report also shows that more than 48% of the total number of operating systems is running Windows XP. Therefore, the attacker will gain more success and larger returns by attacking Windows XP users compared to targeting Linux or Mac operating systems because if we assume that the attacker could infect 0.9% of the total number, then targeting 48% would be much greater than targeting 5% 0r 8% respectively.

5. **Unintended use:** human beings have the tendency to use systems in unintended ways. For example, when the Internet "boom" started, most applications and operating systems were not intended or capable of supporting distributed operations. Despite that, people connected them to the Internet and made them available to users through the network without appropriate levels of security. This will not only expose new vulnerabilities in the system, but also opens doors for remote exploitation.

   Most active attacks are carried out by exploiting existing vulnerabilities in the system. These vulnerabilities are divided into three main categories: 1) architectural design vulnerabilities, 2) implementation vulnerabilities, and 3) operational and platform vulnerabilities [86]. All three categories are typically

Table 1.1. Long term trends of Operating System Usage

2010	Win7	Vista	Win2003	WinXP	W2000	Linux	Mac
October	26.8%	9.9%	1.1%	48.9%	0.3%	4.7%	7.6%
September	24.3%	10.0%	1.1%	51.7%	0.3%	4.6%	7.2%
August	22.3%	10.5%	1.3%	53.1%	0.4%	4.9%	6.7%
July	20.6%	10.9%	1.3%	54.6%	0.4%	4.8%	6.5%
June	19.8%	11.7%	1.3%	54.6%	0.4%	4.8%	6.8%
May	18.9%	12.4%	1.3%	55.3%	0.4%	4.5%	6.7%
April	16.7%	13.2%	1.3%	56.1%	0.5%	4.5%	7.1%
March	14.7%	13.7%	1.4%	57.8%	0.5%	4.5%	6.9%
February	13.0%	14.4%	1.4%	58.4%	0.6%	4.6%	7.1%
January	11.3%	15.4%	1.4%	59.4%	0.6%	4.6%	6.8%

injected during the planning, design, development, deployment, upgrade, or maintenance phases. Human faults made during the design phase result in injecting architectural design errors into the models structure and could prevent the system from functioning securely. Thus, human faults made during the writing of the code during any of the remaining phases would lead to implementation vulnerabilities, which could prevent the system from functioning securely. Consequently, faults in configuration files are operational and are considered platform vulnerabilities. The most dangerous vulnerabilities of all categories are the ones leading to immediate unauthorized access of the application [68, 77, 57, 97]. Providing high levels of application security is paramount in ensuring network, systems, and data security.

1.2. Chapter Conclusion

The introduction of this praxis focused on analyzing application security and the challenges facing its successful implementation. In the second and following chapter, we study the related work done in the field of application security, and the current approaches to reduce or eliminate vulnerabilities from being injected in the code during the software development life cycle. Further, we explain the N-Version Programming Methodology, and its past implementation, and challenges. In the third chapter, we propose the usage of the N-Version Programming Methodology to enhance security by implementing the N-Version Application Security Automation (NVASA) Framework. In addition, we explain the building blocks of the NVASA Framework and investigate the added layer of protection that the NVASA Framework introduces. This layer is capable of detecting and preventing known attacks while having a scheme of protection to zero-day attacks. Additionally, we introduce methods to reduce the overhead associated with the implementation of the NVASA framework. In the fourth chapter, we study Security in Cloud Computing and the related and current work in the field. Moreover, we introduce the usage of the NVASA Framework within cloud computing to enhance the security of the applications running on the cloud and strengthen the resiliency of our framework. In the fifth chapter, we present experimental work to analyze the usage of the NVASA Framework in real-world and actual projects. Furthermore, we emphasize the guidelines and policies that should be followed when building or applying the NVASA framework from our experimental experience. Finally, in the seventh chapter, we conclude this work with a future glance.

Chapter 2

RELATED WORK

2.1. Chapter Introduction

Application security is often one step behind the latest cyber-attack schemes for reasons discussed in the previous section. Bruce Schneier described the race between the white-hat and black-hat community as, *"It's an arms race, and when technology changes, the balance between attacker and defender changes."* [95] The current emphasis on application security is to fix existing implementation errors that could be exploited by publicly known attacks such as Buffer-Over-Flow (BOF), SQL Injection, String Format, and Cross Site Scripting (XSS) [90, 68, 57, 51]. Many of these security schemes are performed during the testing phases of the software development life cycle. Other related work emphasizes extensive revision of design and source code during and after implementation. Furthermore, many schemes were developed to test the security during the quality assurance phase like white-box testing, black-box testing, unit testing and many more. Moreover, runtime checks were introduced for safety-critical systems to eliminate or reduce the effect of errors during the runtime of the application. Finally, penetration testing is performed on the final live product periodically to make sure that any misconfiguration, out-dated software, and/or unpatched applications are discovered early. Thus, prioritizing the security issues to be fixed [89, 29].

In this chapter, we will discuss these schemes and their shortcomings in detecting all application security. Additionally, we will look at the top publicly known applica-

tion vulnerabilities and their negative impact on organizations and companies.

2.2. Application Security

Most current related work in the field of application security focuses on enforcing extensive security guidelines during the software development life cycle process [53, 79]. These guidelines focus on providing the regulations needed to promote the design, development, deployment, upgrade, and maintenance of secure applications throughout their lifecycle. The initial approach to application security espoused manual and automatic audits to the source code then during production unit testing is a method of testing each unit and finally after integrating all system components testing is the method to verify that the application is bug-free and that it functions correctly.

2.2.1. Code Review

Code review approach consists of reviewing and reading the source code manually line by line to detect existing vulnerabilities that was injected during the design and development phases. Then reporting these vulnerabilities to be verified and fixed; therefore, writing less buggy code and more reliable applications. This approach is done without executing the application.

There are three kinds of code review; first, heavyweight meeting based inspection. Second, the lightweight review process. Third, pair-programming [69]. The heavyweight inspection is based on scheduled meetings, paper-based code reading and tedious metrics-gathering. Lightweight code review drops the meetings and engages in less overhead and less formal code inspection. Finally, the pair-programming (Extreme-programming) [60] where two programmers work together; one programmer writes the code while the other perform revision of written code on the spot. They

alternate based on a specific criterion.

2.2.2. Static Code Analysis

Static code analysis approach is performed on the source code without executing the application as well. Static code analysis consists of reviewing the source code via specific tools and software; therefore, automating the review process to detect existing vulnerabilities and report those for verification then send them to be fixed [59]. In static code analysis, there are many methods such as model checking and data-flow analysis and assertions, etc.

2.2.3. Code Testing

Code testing is one of the critical phases of the SDLC and is widely used for performance, usability, integration, and security testing nowadays. The most popular testing methods include white-box, black-box, unit, dynamic and static testing of the source code. For application security Fuzzing testing is a method used to test user input vulnerabilities. "Fuzzing" is usually performed as a black-box testing but could also be performed as a white-box testing as well. "Fuzzing" is followed by inputting a distorted or illegal input, usually invalid, unexpected, or random input to the application and monitoring the behavior of the application [101] to report any crashes or unanticipated behavior. Sulley and SPIKE frameworks [102] are two examples of "Fuzzing" testing. Another application security testing method is boundary testing [78], where the conditions are defined to generate extreme test cases. These boundary cases are generated from the edges of the input pool. These boundary test cases are subsequently tested against the application. Accordingly, the behavior and results are monitored for errors, crashes or unexpected behavior to be reported and then hopefully fixed.

2.2.4. Runtime Check

Runtime and compile-time checking; are two methods for detecting and preventing the exploitation of existing vulnerabilities. These methods are followed by adding special checks within the source code to ensure the program behaves as desired. ProPolice framework [72] is a GNU Compiler Collection (GCC) extension developed by IBM to protect against stack smashing that uses runtime checking. Mudflap framework [71] is another GCC extension for pointer debugging that uses runtime checking to detect and prevent exploitation of vulnerabilities.

Some computer languages like Ada, Eiffel, and OCamel are equipped with runtime checks capabilities. In Ada computer language, runtime checks can detect buffer-overflow, access to unallocated memory, and array access errors as well as many other detectable errors. Ada is a widely used computer language in military and missile control systems [1, 55].

2.2.5. Penetration Testing

Penetration testing is a testing scheme, commonly associated with security testing. The main focus in pen-testing is to attempt to gain un-authorized access to networks, servers, or applications without any knowledge of usernames or passwords [89]. Penetration testing could be performed in black-box or white-box fashion. This is usually decided by management based on the architecture of the network, and the compliances required. It is very important to understand that a pen-tester will not find all vulnerabilities. Consequently, there is no guarantee that the system is %100 secure even after a pen-test.

Guidelines, Security testing, code review, static code analysis, and runtime checks are important components of an applications software development life cycle, but these solutions by themselves are limited and have major disadvantages if we rely on

them solely. First, actual security vulnerabilities are triggered by a certain specific set of circumstances, which makes it extremely hard to automate the testing of the application. For example, it is hard to exploit an application using random "fuzzing" testing. A second limitation is the considerable investment of resource and time needed to perform manual code revision. A third limitation is latent security vulnerabilities that are present within critical systems but concealed from the systems stakeholders. These latent vulnerabilities can damage an organizations reputation and could lead to financial loss. Further, it takes an average of 14 weeks to patch or fix an existing vulnerability after discovery, which opens a window for additional attacks leading to more damages [75].

A fourth limitation is the high cost of implementing a secure application because the enforcement of security training as well as hiring special security testers and purchasing specific security tools adds enormously to the total cost. A fifth limitation is the increased time-to-market since these solutions engage in comprehensive training and thorough testing of the code. Sixth, following these extensive guidelines often results in the detection of known existing vulnerabilities. However, these extensive guidelines have no scheme for detecting or preventing latent or zero-day vulnerabilities from being exploited. Seventh, runtime check schemes are usually commonly to a certain vulnerability [11, 72, 71]. Thus, to secure an application from hundreds of vulnerabilities, it would require the usage of many run time schemes, tools, libraries and packages to protect against every single one of them. Moreover, Run time checks are regularly criticized of slowing down the implemented application; therefore, nearly all compilers enable the developers to switch off this feature if performance is a concern, which is typically the case with most enterprise applications.

Finally, penetration testing is an excellent technique to discover known-vulnerabilities. However, pen-testing has few disadvantages. The first disadvantage is that it requires

an expert in the field to perform an effective penetration testing. In many cases, it is performed by contractors, which is expensive and needs to be executed periodically. A second disadvantage is the false sense of security it gives to management between pen-test rounds. This is a serious issue because if a pen-test was performed today on a particular network segment, and tomorrow few patches were committed to Microsoft or Unix servers; these patches could introduce new vulnerabilities to the network segment. Therefore, will no longer be 'secure'. However, the management would falsely believe that the segment is secure. In addition, a third disadvantage is that pen-testing focuses on discovering known vulnerabilities. Testers examine systems for specific patterns, software versions, and OS types to report known vulnerabilities. Hence, falling short in detecting and preventing zero-day attacks.

2.2.6. Application Security Attack Phases and Countermeasures

A typical application attack in most cases would take four phases to complete as shown in Figure 2.1 The first phase, the attacker scans the server and all applications running on the server to learn the host machines operating system (OS) as well as the software and tools running within. This knowledge would enable the attacker to learn what vulnerabilities to search for based on the OS type/version as well as the applications version running on the server. For example, if the attacker learned that the server is running Windows based OS he/she would greatly benefit from learning which IIS version the server is running. The second phase, the attacker identifies the applications running on the server. Therefore, identifying these versions and the vulnerabilities they could suffer from. Thus, launching a series of targeted attacks to exploit the vulnerabilities (Known to the server or applications version discovered) and gain unauthorized access to the machine or the application. Then in the third phase after finding the vulnerability and succeeding in exploiting it to gain

unauthorized access the attacker would eavesdrop, steals, damage or tamper with the victims confidential data. The fourth and final stage, the attacker covers his tracks by deleting the evidence from the OS system log files as well as any other tracking log files within the victims private network. Log files such as intrusion detection and firewall information.

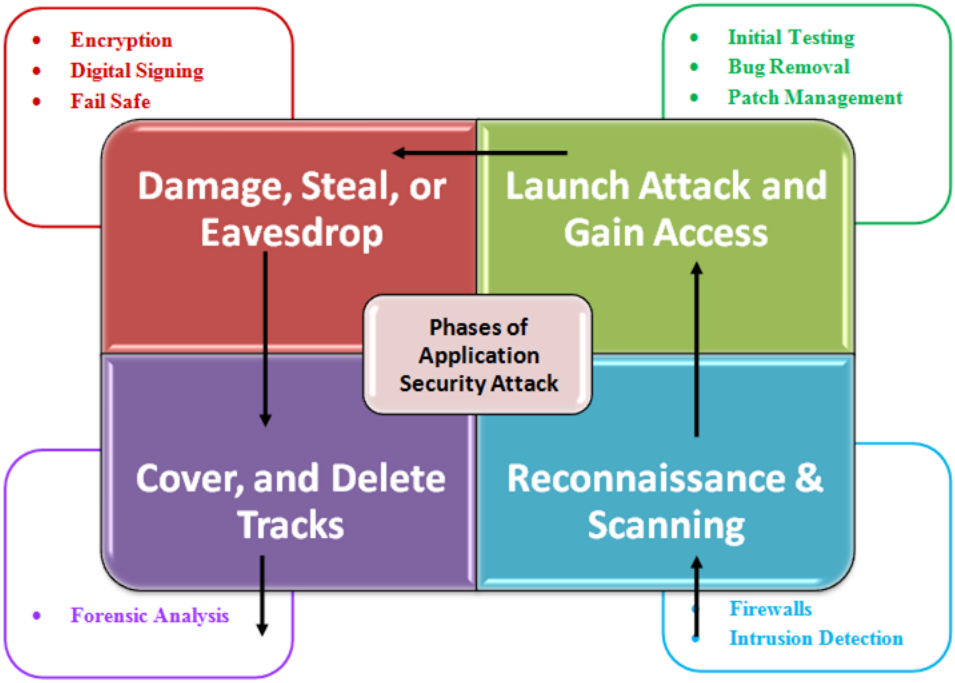

Figure 2.1. Application Attack Phases and Countermeasures

Each of the four phases has countermeasure techniques that are being implemented within the Internet and Intranet today. Figure 2.1 Shows the most commonly used techniques in clear text boxes beside each phase.

2.2.7. Top Known Application Security Attacks and Current Mitigation schemes

As mentioned previously, vulnerabilities fall into three categories. We will investigate some of the leading known vulnerabilities published by SANS Institute, MITRE, and many top software security experts in the US and Europeand [64, 112]. In addition, we will discuss the current mitigation schemes and some of their disadvantages.

2.2.7.1. SQL Injection Attack

SQL Injection attacks are performed by injecting a crafted text-based input to exploit the application's interpreter [64, 112, 57, 40]. A simple SQL injection example is having a web server query a username $result=mysql_query('SELECT * FROM users WHERE username='".$_GET['username']."'"); [31]. Where an attacker inputs a SQL query with 'username' such as ' or 1=1#; therefore, bypassing the authentication page.

There are few techniques to mitigate SQL injection. The first and widely used technique is validating the external input before applying the query. This is achieved by checking for the type, length, format and range of the input before passing it on [22]. Unfortunately, some language libraries are vulnerable to certain inputs and can be bypassed using special characters or formats. For example, using backslashes or comment tags [39]. A second technique, is using parameterized queries with bound parameters to separate the query from the input data [22, 32]. However, some hackers managed to bypass this protection and inject the database using a more complex injection scheme shown by Tylor, C. [38]. A third effective technique to mitigate most SQL injection attacks is using stored procedure [41], but the implementation of the procedure must be performed by experts to guarantee a correct implementation. Some implementations have proven to be vulnerable to SQL injection [32]. In addition, store procedures are usually criticized for being vendor-specific [42], switching to another database most likely would require a new development of the store procedure.

2.2.7.2. OS Command Injection Attack

This attack happens when the application allows untrusted input to be executed by the OS to run a section of the application or an external program [64, 77]. An attacker can take advantage of this bridge and run OS malicious commands. A simple vulnerable C function is shown below. The function takes a filename and displays the content to the user. If an attacker injects a simple command ";rm -rf /" instead of the filename, he/she will be able to delete the content of the residing folder on a Unix box. To mitigate this type of attack, input validation must be performed before executing the command on an OS level. Unfortunately, some attacks were reported to bypass the filtration schemes using complex input formats [36, 62].

```
int main(char* argc, char** argv) {
                char cmd[CMD_MAX] = "/usr/bin/cat ";
                strcat(cmd, argv[1]);
                system(cmd);
        }
```

2.2.7.3. Classic Buffer Overflow Attack

Buffer-over-flow attacks as previously explained happens when a buffer boundary is exceeded [64, 68, 90, 112], this could enable an attacker to overwrite the return pointer to run malicious code on the hosting server. Here is a simple example [9].

```
void example(char *s) {
  char buf[1024];
  strcpy(buf, s);
}
int main(int argc, char **argv) {
  example(argv[1]);
}
```

The most widely used scheme to mitigate BOF vulnerabilities is by forbidding the usage of vulnerable functions during the development of the application. One

example is the usage of the 'banned.h' header file developed by Microsoft to support the SDL requirement. This file basically flags a warning when a dangerous function or library is used by the developer [7, 8]. Unfortunately, developers still write unsecured code using some of the known vulnerable functions and libraries [64]. In addition, in some cases legacy code is being re-used without any revision or testing.

Another technique is using canaries which are known values placed between a control-data and a buffer to monitor for BOF. Therefore, if a BOF occurs, the canaries will be the first to be affected [11]. Unfortunately, canaries cannot protect from all BOF types. For example, there is no protection against over-flowing the heap [17]. In addition, protection schemes must account for each attack vector, given that there is no environment protection throughout the execution.

2.2.7.4. Upload of Dangerous File Types Attack

Some websites allow uploading files to their hosting systems for processing and/or storage [64, 48, 49]. An attacker can upload dangerous types of files through the application in order to run malicious code on the hosting servers. A simple vulnerable PHP website is demonstrated here.

HTML code:

```
<form enctype="multipart/form-data" action="uploader.php" method="POST">
<input type="hidden" name="MAX_FILE_SIZE" value="100000" />
Choose a file to upload: <input name="uploadedfile" type="file" /><br />
<input type="submit" value="Upload File" />
</form>
```

Corresponding PHP code:

```
<?php
\$target_path = "uploads/";
\$target_path = \$target_path . basename( \$_FILES['uploadedfile']['name']);
if(move_uploaded_file(\$_FILES['uploadedfile']['tmp_name'], \$target_path)) {
echo "The file ". basename( \$_FILES['uploadedfile']['name']).
```

```
" has been uploaded";
} else{
echo "There was an error uploading the file, please try again!";
}
?>
```

In this example, there are no restrictions on the uploaded file, in most cases a simple whitelist on the file name is performed, which can be easily bypassed [30]. Other widely used technique is performing an anti-virus scan before accepting/storing the file [48]. However, bypassing a single anti-virus scanner is not a difficult task. Hackers have developed schemes to overcome detection by these anti-viruses. For example, hackers commonly change the payload of the malicious file, or encode the executable to bypass most anti-virus scanners undetected [3].

2.2.7.5. Cross Site Scripting (XSS) Attack

XSS is another attack performed by injecting client-side script that exploits the trust that the client has in the web server [14, 15]. A severe XSS attack can be performed by injecting a malicious script in a forum post page that resides on the web server that runs on the visitor's machine to steal the session cookie and send it to the attacker.

XSS attack is slightly different than the rest. It exploits two aspects of the connection. The first, the server-side to harm users and administrators. This is achieved by storing malicious scripts within the server. This can happen mainly for two reasons, either that the website does not validate the input or by crafting the input to bypass any filtration schemes performed by the hosting site. The second, the client/user-side. This is achieved by running scripts on the client's web browser (IE6.0, FireFox, Safari, etc.). Some browsers are vulnerable to specific scripts while others are immune to them, but they could be vulnerable to another set of scripts [13].

Mitigation schemes focus on escaping and filtering any external input. Escaping data is achieved by ensuring that the input characters are treated as data, not as the interpreter's characters [15, 14]. Again, it has been shown that specific complex inputs can bypass these techniques [13].

2.3. The Methodology of N-Version Programming (NVP)

The concept N-Version Programming was introduced in the late 1970s by Liming Chen and Algirdas Avizienis as a fault tolerance method to improve the reliability of the software operation. NVP is defined as *"The independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification."* [58].

N-Version programming methodology is introduced by the following central hypothesis: *"The independence of programming efforts will greatly reduce the probability of identical software faults occurring in two or more versions of the program."* [58].

The strength of the NVP stems from the diverse versions of the same function as well as the decision algorithm. The decision algorithm simply looks for a consensus based on the received outputs. In some cases, the NVP decision algorithm was used with floating-point numbers by taking the median of the outputs instead of a majority output [99].

Building unique and diverse N-Versions is essential to obtain the reliability enhancement introduced by Liming Chen et al. It is also important to synchronize the versions and solve any timing constraints that may be introduced by the diversity in language, algorithm and data structure. Thus, to build a pure N-Version model as shown in Figure 2.2 three policies must be adopted:

1. All versions must share the exact same initial specification. The purpose of the initial specification is to state the functional requirements that stakeholders want the application to perform. These initial specifications must be precise,

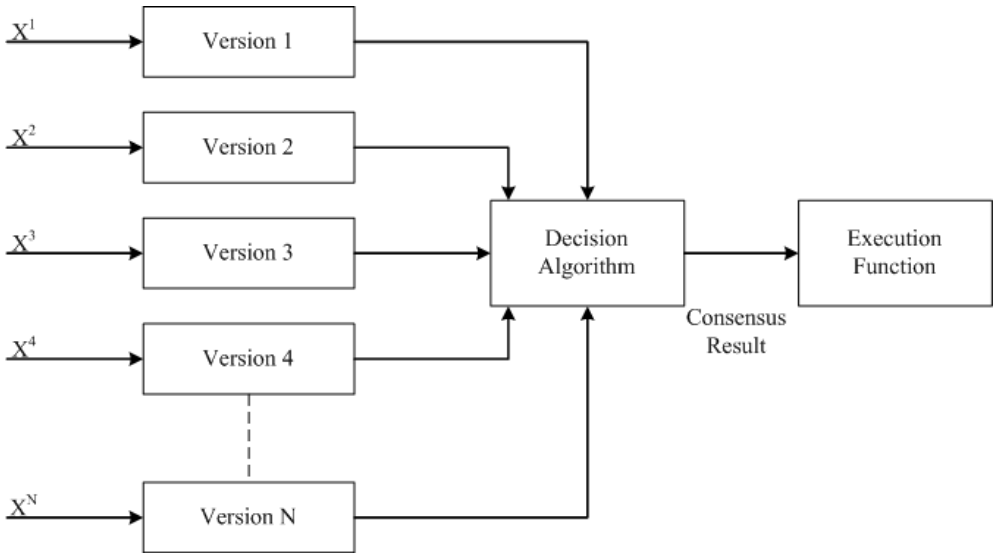

Figure 2.2. N-Version Model [58]

clear and detailed oriented to eliminate any confusion during the development process. In case the N-Versions output wasn't precisely comparable, a translator must be implemented before a consensus algorithm is performed to successfully compare the N-Versions' output. As a result, reach a majority or a unanimous output.

2. Versions must be independently generated. This is achieved by choosing different algorithms and programming languages for each version, as well as the independent processes for generating the versions, which should be carried by N independent individuals or groups that have no interaction with each other. This isolation of design and process between groups, coupled with the diversity of choosing programming languages and algorithms, greatly reduces the probability of producing identical software faults in two or more versions [58, 82].

3. Versions must run concurrently to eliminate or reduce any delay correlated with the usage of the N-Version programming methodology. In addition, any ver-

sion synchronization or timing restraints must be administered at the decision algorithm level.

2.4. Chapter Conclusion

In this Chapter, we have discussed the common existing methods of building and maintaining secure applications. Some of which are formal, or informal code revisions. Where the focus is on revising the code manually and/or automatically. Other methods involve special testing schemes during the development phases. In addition, penetration testing performed on the final production system. Further methods involve Runtime and compile-time checks, which add special source code checks to ensure that the program functions as planned. All these methods are effective in detecting some of the known-attacks, but they all suffer from unique weaknesses mentioned in the chapter. Most importantly, all these methods fall short in detecting zero-day exploits.

We then followed our discussion to explain the N-Version Programming Methodology which was introduced by Liming Chen and Algirdas Avizienis to imporve hardware relaibility. However, the N-Version programming methodology is usually criticized for having high resource requirements, overhead, and implementation time. In addition, some specialists argue that errors in some cases are correlated and can be equivalent in two or more versions, even though the versions were developed independently [63]. On the contrary, most experiments done in the field proved that the errors injected during the development of the versions are not correlated and that the probability of having two or more identical errors in two or more versions is immensely reduced.

Due to the costly overhead needed to develop the N-Versions; implementation of the methodology was limited to safety-critical systems as well as systems with a high

reliability requirement. National Aeronautics and Space Administration (NASA) is one example where the N-Version programming methodology was used in their space missions heavily to mitigate bugs and increase the overall reliability of their systems [80]. Another implementation of the N-Version methodology was in train switching control systems, where safety is very critical to their passengers, cargo, and the cities along their rout.

In the next Chapter, we introduce the usage of the N-Version programming methodology to build the NVASA Framework to moderate programming language, algorithm, data structure, compiler errors, and security vulnerabilities provoked during the SDLC. We later investigate the added overhead associated with the NVASA framework and discuss schemes to minimize this added overhead, while leveraging the enhanced security introduced by the NVASA framework.

Chapter 3

N-VERSIONS ARCHITECTURAL FRAMEWORK FOR APPLICATION SECURITY AUTOMATION (NVASA)

3.1. Chapter Introduction

Our proposed architecture introduces the N-Version programming methodology to produce an architectural framework for application security automation. This framework is capable of detecting most known attacks while having a scheme of protection to detect and thwart unknown attacks as well [83]. Our main goal here, is to achieve better overall system security while reducing security efforts in all the design, development, deployment, upgrade, and maintenance phases.

To build the NVASA framework, we first collect the initial specifications and requirements from stakeholders and feed them to the NVASA Automatic Generator. The NVASA Automatic Generator then generates the NVASA framework with the N-Versions outline. Each version outline is written in a different programming language specified by the initiator. The N-Version outlines are then handed to the N-Groups of developers for further development. Figure 3.1 shows the process of building the NVASA framework.

However, the NVASA framework adds overhead to the development phase in the Software Development Life Cycle (SDLC) compared to a single version implementation. Therefore, we explore methods to reduce the overhead associated with the NVASA framework implementation. We discuss in detail compartmentalizing the application's architecture into two or more components to identify the critical com-

ponents through a security evaluation. Consequently, apply the NVASA framework to the critical components. In addition, we discuss other methods to reduce overhead through cross-platform compilers and the usage of source-to-source language translators to achieve some diversity.

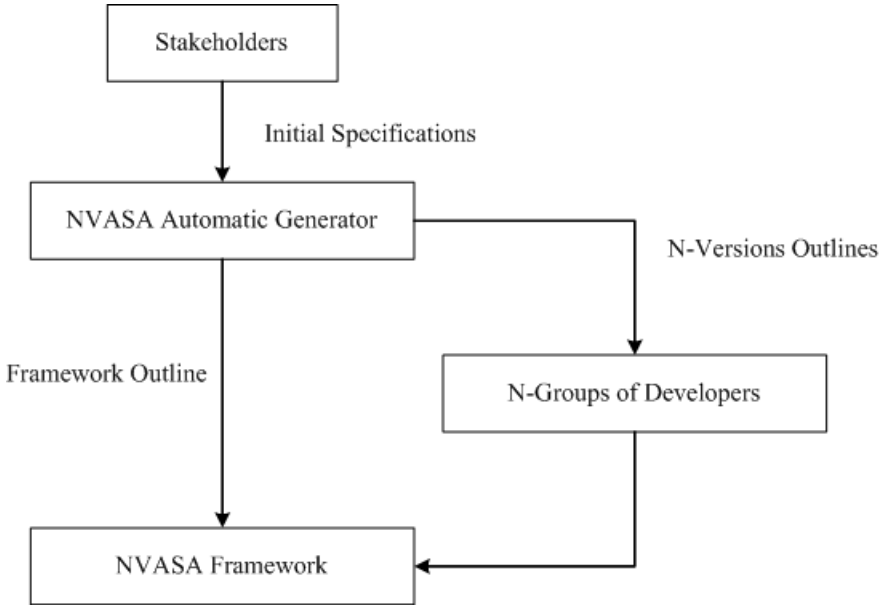

Figure 3.1. NVASA Framework Building Process

3.2. NVASA Building Blocks

NVASA framework is constructed of four layers as shown in Figure 3.2 The first layer is the N-Version routing layer. The second layer is the N-Version environment layer. The third layer is the N-Version decision layer, and the fourth layer is the backend application server layer.

3.2.1. N-Version Routing Layer

The N-Version routing layer is the interface of the application to the outside world; it connects the framework with users/applications over the network. It is responsible for receiving requests from external users and routing these requests to the N-Versions in the environment layer to be executed. The routing layer is also responsible for replying to requesters with the appropriate response if and only if their request was legitimate after executing the request; otherwise, reply with an error message.

3.2.2. N-Version Environment Layer

The N-Version environment layer contains the N-Version applications where each version is designed and developed by an independent individual or group using a unique algorithm and/or programming language as mentioned in refch:relatedwork Each version receives its execution command along with the identical input of all N-Versions from the routing layer. All the versions then execute concurrently, which eliminates any reduction in performance. The N-Versions can be either coupled running on diverse platforms on different physical locations or tightly coupled running on the same physical machine. This is decided based on the initial design specification of the NVASA framework.

Loosely coupled versions have many advantages: 1) The time overhead is reduced compared to tightly coupled versions running on one machine. 2) There is no single point of failure within the N-Version environment layer compared to tightly coupled versions. 3) Loosely coupled systems scale better than tightly-coupled systems. On the other hand, loosely coupled versions add more complexity to the development and testing phases of an application since messaging schemes must be implemented to connect layers and components. Additionally, the communication channels need to be protected through transport or network layer security protocols. Depending on the type of applications, communication between various layers and versions may

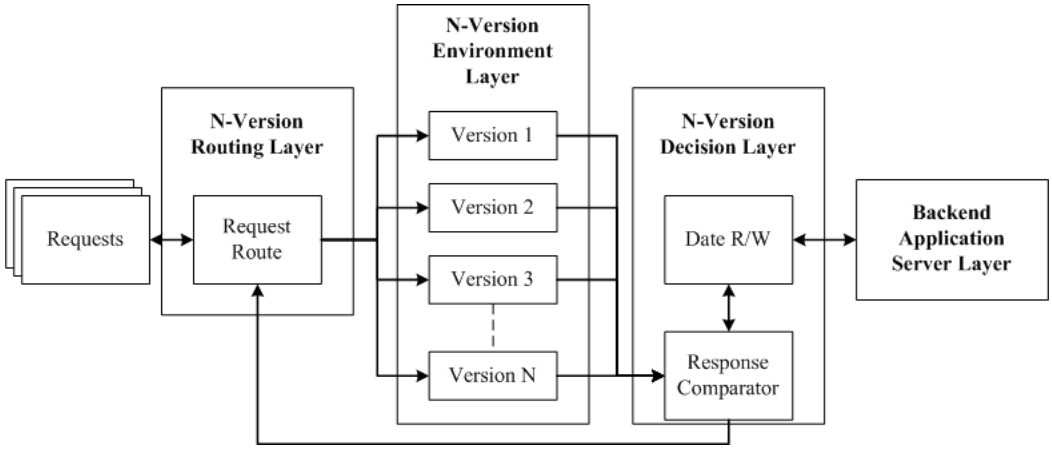

Figure 3.2. NVASA Framework Four Layers [83]

cause additional time overhead.

3.2.3. N-Version Decision Layer

The N-Version decision layer is the brains and intelligence of the NVASA Framework. The N-Version Decision Layer is composed of two components: 1) The Response Comparator component and 2) The Data Read and Write (R/W) Component. The response comparator component receives the N-Version outputs. Then the decision algorithm applies a generic consensus rule to determine the majority output as shown in Figure 2.2. The responsibility of the decision algorithm is to determine exploited versions by identifying contradicting output compared to the majority of the versions and removing the exploited and breached versions from the pool, thereby eliminating any malicious attempt to exploit the application by preventing the conflicting output from being executed or performed on the backend databases and servers.

The decision algorithm simply looks for a consensus based on a majority output or takes the median if the output is a floating-point number [99]. If necessary the

response comparator component then passes the consensus output to the data read and write (R/W) component to generate the R/W command from the consensus output to be applied to the backend application server layer. Finally, the data R/W component generates the output or confirmation based on the initial request and passes it to the response comparator to be then sent to the request route component to reply to the requester.

3.2.4. N-Version Backend Application Layer

For applications that read or write data to/from a server or database, the backend application server layer is essential. The N-Version Backend Application Layer holds the actual servers and/or databases that the application uses. This layer receives the agreed output, if attained, from the data R/W component from the N-Version Decision Layer. This prevents any direct communications between the backend application server layer and the N-Versions in the environment layer. This significant design requirement prevents any successful exploit(s) from propagating further within the system. Therefore, preventing any vicious exploit from modifying or leaking data included in the backend application server layer.

3.3. Reduction Techniques In the NVASA Framework

3.3.1. Compartmentalization

With respect to the publicly known software architecture styles, here we are going to study the commonly used architecture styles that use compartmentalization to separate the application into two or more components. In addition to the benefits added by using styles that offer compartmentalization like preventing failures and security attacks from propagating through the entire system; these architecture styles

facilitate the usage of the NVASA Framework [83] by versioning only the critical components rather than applying the NVASA Framework to the entire application. By applying the NVASA Framework to the security critical components within an architecture, we reduce the total cost and effort spent during the SDLC compared to applying the NVASA Framework to the entire application.

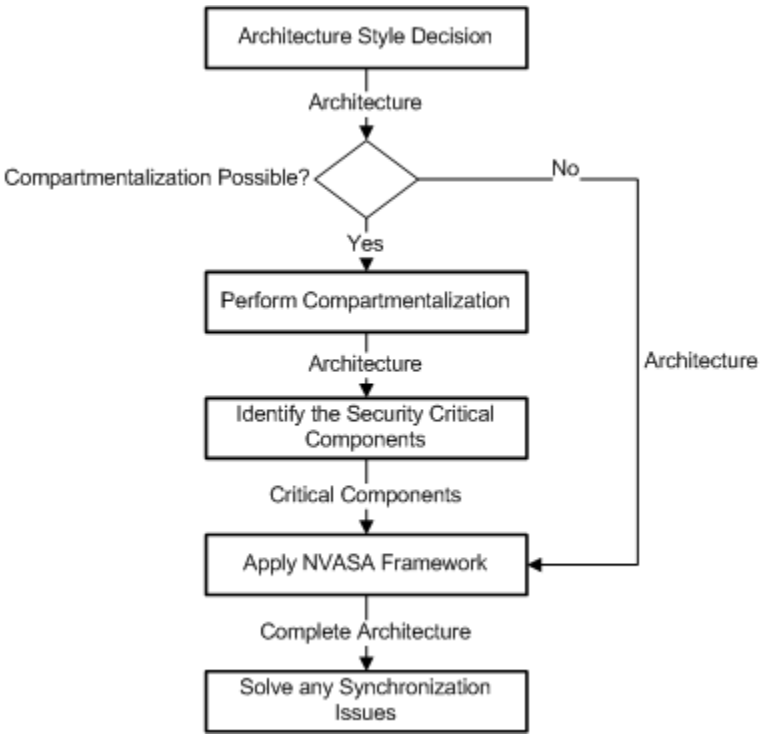

Figure 3.3. Decision Flow

In order to design secure software, few decisions must take place regarding the software architecture during the requirement gathering, and analysis phases. These decisions would enable the software architecture to take advantage of the NVASA Framework in two different ways. As seen in Figure 3.3, a decision must be made on which style to use. Based on the style selected it will determine whether the NVASA Framework can be implemented on the critical components or the entire application.

If the architecture style does not support compartmentalization then the only way is to apply the NVASA Framework to the entire application as proposed in [83]. Otherwise, if compartmentalization is a possibility then a security evaluation must take place to identify the critical components, we will explore two methods to identify the critical components later in this chapter. Finally, the critical components are then fed to the NVASA Automatic generator (Generator Figure) to spit out the N-Version templates to be handed to the N-Group of developers for further implementation.

The simplest style is the main program and subroutines architecture which emerged from the usage of computer languages such as C, C++, Java and Pascal. This style can be used to implement different architectures that we will investigate later in this paper but the basic form of the main program and subroutine style is not feasible for compartmentalization due to the limitation usage of one programming language; therefore, N-Versioning the entire program as the initial NVASA Framework paper proposed is the only way to benefit from the NVASA Framework in terms of security.

Another popular style is the ObjectOriented (O-O) style [70]. O-O is very similar to the main program and subroutine the only structural difference is that the O-O style uses objects, which enable encapsulation, polymorphism, etc. O-O style can be compartmentalized as shown in Figure 3.4. Therefore, identifying the critical components In addition, N-Versioning these critical components is possible and would enhance the application's security if designed properly.

Pipe-and-filter is another data flow style that is very robust and popularly used [74]. This style can consist of any number of filters that percolates data based on pre-defined rules before passing it to the next filter using pipes (Connectors). The pipe-and-filter style is widely used in a sequential fashion. However, it can be used in a more complex, and parallel structure. Figure 3.5 shows how the NVASA framework

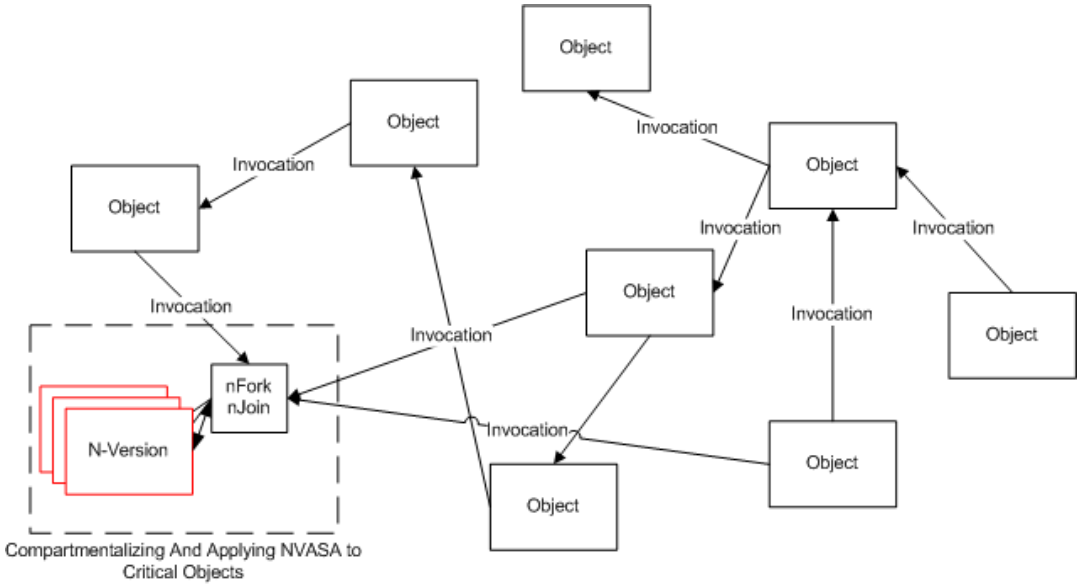

Figure 3.4. Applying NVASA to Object-Oriented Architecture Style

would apply to a critical filter.

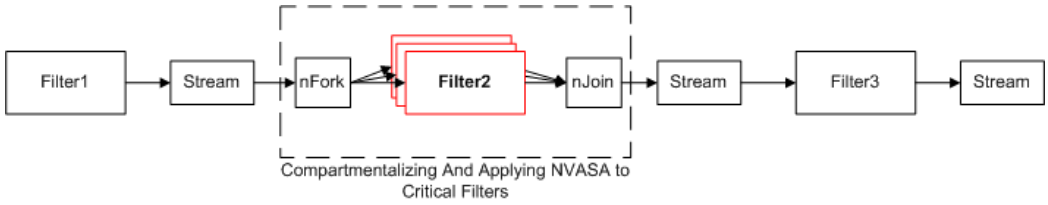

Figure 3.5. Applying NVASA to Pipe-and-Filter Architecture Style

Virtual machine style is a layered style, simple in architecture and widely used [87]. The fundamental nature of this architecture is the separation of programs into layers, where each program is permitted to obtain services from the layer below it. The layered style can benefit from the NVASA framework by identifying the critical programs within the layers or possibly identifying the critical layers. Then based on the approach selected, the NVASA Framework is applied to either the critical

programs within the layers or the entire layer as shown in Figure 3.6. Therefore, reducing the risk of exploiting these critical programs, or layers. Defining the security critical programs, and/or layers is essential and must be performed accurately.

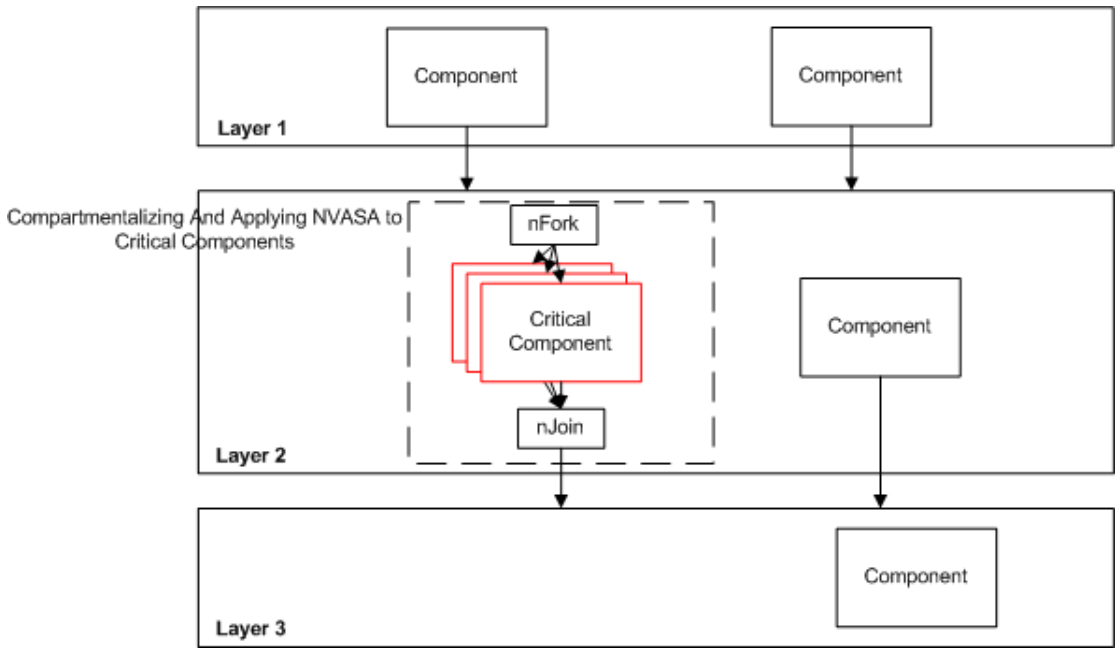

Figure 3.6. Applying NVASA to Virtual-Machine Architecture Style

Blackboard architecture style merged merely to serve artificial intelligence applications. Blackboard style was defined by the idea of many diverse experts gathering around a blackboard cooperating in the solution of a large and complex problem. The blackboard style can benefit from the NVASA framework by identifying the critical expert programs then using the NVASA Framework as shown in Figure 3.7 to N-Version these expert programs to reduce or remove the risk of exploiting them. Defining the critical expert programs within the blackboard architecture style is essential and must be performed correctly.

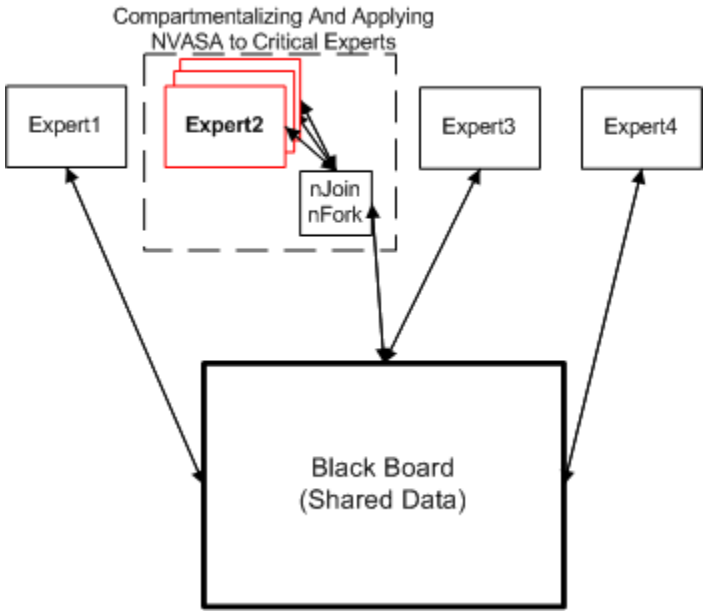

Figure 3.7. Applying NVASA to the Blackboard Architecture Style

3.3.1.1. Identifying the Critical Components

In order to reduce the NVASA framework effort and overhead; we propose first identifying the components with the highest risk of exploitation. Consequently, applying the NVASA framework exclusively to the identified crucial components to improve the security of the entire system. We propose two ways to identify the critical and sensitive components. The first is *Static Analysis* using automated parsing methods. The second is *Functionality Knowledge*, through security architecture experts.

Static Analysis

A simple and automated way to identify sensitive components within an application is by parsing through the source code and looking for input/output (I/O) commands, database query, and file access. Essentially, any input source, whether from a remote application or user. These lines of code usually carry most of the application's vulnerabilities [64]. Therefore, applying the NVASA framework to the

functions and components carrying these commands would increase the system's security while minimizing the overhead associated with the application of the NVASA framework. We explore each critical component separately.

User Input (I/O)

User input is considered a critical part because an attacker can craft an input in an un-expected form by the application [64]. Therefore, we consider the user input code as well as any filtration code to be critical. Parsing through the source code we can search for input command based on the language used. For example, if the application was written in C++, we will be parsing the source code for " cin<< ". Another example is "<input" in the PHP language where the input can be one of the following HTML forms: text field, password field, radio button, check box, or submit button [23]. Therefore, applying the NVASA framework to the first line of functions would assure a sanitized output to the next layer of functions preventing any exploits from propagating and executing.

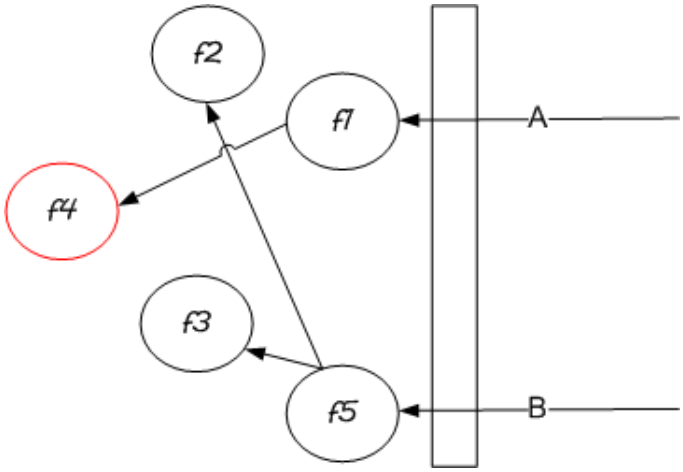

Figure 3.8. Simple Web Architecture

Figure 3.8 shows a simple web architecture with *f1* and *f5* as the interface functions, assuming that *f4* was vulnerable to a buffer-over-flow, if an exploit was attempted through *f1* it could possibly propagate to *f4* and exploit it. Therefore, applying the NVASA framework to the first line of functions as shown in Figure 3.9, and restricting any buffer following the path of *f1* and *f5* to be equal to or larger than the buffer within the first line of functions. Therefore,*f1* and *f5* will guarantee a sanitized output to *f4* *f3*, and *f2* through the decision algorithm. Thus, detecting and preventing the request from propagating if a consensus was not reached.

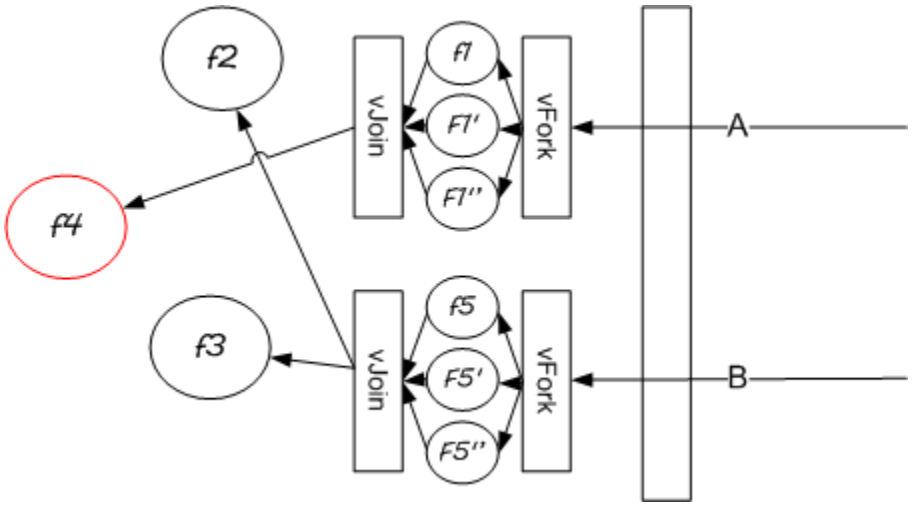

Figure 3.9. Applying the NVASA Framework to the Simple Web Architecture

Database and Database Query

Applications that dynamically generate SQL queries based on user input and does not perform any or poor input sanitization are vulnerable to SQL injection [64]. This is considered a severe vulnerability because an attacker can modify the structure of the query forcing the database to perform un-intended functions. Parsing through the source code we can search for SQL query command lines, for example, parsing the source code for “SELECT * ”, “ FROM ” and “ WHERE ” or look for

specific language commands such as the Java load JBBC driver " Class.forName("com.mysql.jdbc.Driver").newInstance(); ". Alternatively, "mysql_" functions in the PHP language. Then apply the NVASA framework to the functions that create and apply these queries. Figure 3.10 shows a model of this approach.

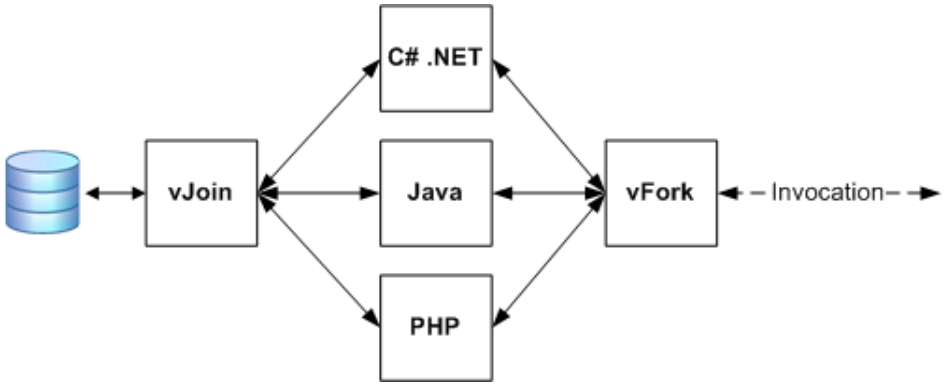

Figure 3.10. Applying NVASA to the Database Component

Another significant approach to secure the database is by N-Versioning the database itself as shown in Figure 3.11. This will prove effective when an attacker crafts a SQL injection that is specific to a database(s) model and version; consequently, the exploit will be successful in exploiting one or more N-Version databases, at least one of the databases will discover the malformed/attack query and not commit it. As a result, the vJoin decision algorithm will get at least one 'false' or 'error' message. Thus, be able to identify that the query is maformed or malicious. Thereby, rollback the transaction [35, 34] of all databases to revert to a safe state.

An example of a SQL injection that is effective only with Microsoft SQL server is "SELECT * FROM members; DROP members−−" [40]. If we are running N-Versions of the database, and one was a Microsoft SQL Server it will be exploited while the rest of the versions would detect the malformed query and reply with an

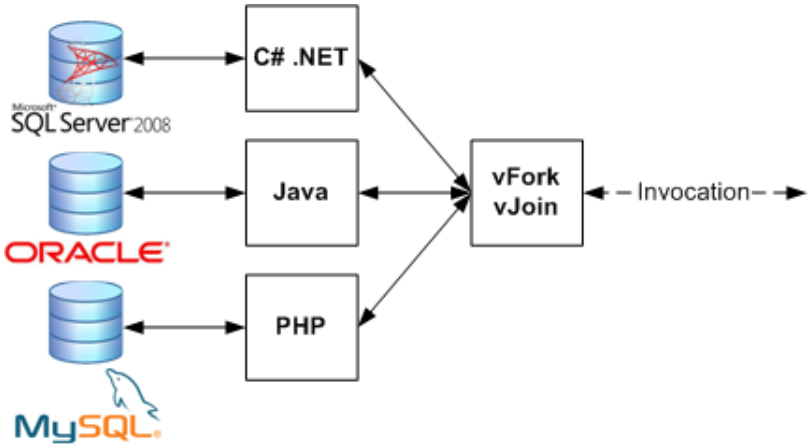

Figure 3.11. Applying NVASA to Databases

'Error'. Thus, the vJoin would receive one or multiple 'Error' replies. As a result, forcing the Microsoft SQL Server to revert to the state prior to applying the query. Table 3.1 shows examples of SQL injection queries and the corresponding exploited databases [64, 40]. A ✗ mark indicates that this specific query was successful in exploiting the database model/version. The point here is that at least one database model will detect and prevent the query. Therefore, enabling the decision algorithm to detect the malicious query, since the algorithm is looking for a unanimous output. Hence, revert the rest and prevent the query from propagating through the system.

Functionality Knowledge

Identifying the critical components through functionality knowledge is a research problem by itself and there are standards produced to address such an issue. This process should be performed by a security evaluator expert who has sufficient knowledge of the application's functionality and architecture due to the complexity nature of discovering potential zero-day vulnerabilities and the hidden security risks within the application's inputs and outputs (I/O).

Table 3.1. SQL Injection Queries and Corresponding Exploited Databases

SQL Injection	**Microsoft SQL**	**MySQL**	**Oracle**	**PostgreSQL**
DROP sampletable;−−	✗	✗	✓	✓
DROP sampletable;#	✓	✗	✓	✓
SELECT * FROM members; DROP members−−	✗	✓	✓	✓
SELECT CHAR(0x66)	✗	✓	✓	✓
CONCAT(str1, str2, str3, ...)	✓	✗	✓	✓
SELECT LOAD_FILE(0x633A5C626F6-F742E696E69)	✓	✗	✓	✓
' UNION SELECT 1, 'anotheruser', 'doesnt matter', 1−−	✗	✗	✓	✓
;shutdown −−	✗	✓	✓	✓
WAITFOR DELAY '0:0:0.51'	✗	✓	✓	✓
SELECT ASCII('a')	✗	✗	✓	✗
SELECT pg_sleep(10);	✓	✓	✓	✗
SELECT login ||'−' ||password FROM members	✓	✗	✗	✓
INSERT INTO members(id, user, pass) VALUES(1, "+SUBSTRING(@@version,1,10) ,10)	✗	✗	✓	✓

There are standards and methods to identify the critical components. For example, The Common Criteria Information Technology Security Evaluation [67] (Common Criteria, 1999) is an international standard widely used to identify the critical security components. Other work has been done by Young, 1991 Verifiable Computer Security and Hardware: Issues by prioritizing the components based on some. Furthermore, other work has been done by Andrew Rae, and Colin Fidge [92] to improve Young's approach by making it more efficient. We adopted Andrew's approach [92] to identify the security critical components within an architecture.

Andrew's approach relies on the category of information manipulated by the component to prioritize and identify the critical components. The assumption here is that the application architecture model consists of components and connections. These connections carry the information from one component to another while the components can generate or manipulate the information to achieve the application's goal.

Based on the carried information, components and connections are categorized into two categories. First, 'Data' Connection/Component where classified information is carried or manipulated by the 'Data' component/connection. Classified data is defined as critical and sensitive data to the application or external world. Second, 'Control' Connection/Component where non-classified information is carried by the 'Control' connection/component. Unclassified information is defined as non-critical data to the application or external world. If a component/connection act as both 'Data' and 'Control', then we classify it as 'data' to protect the classified information within.

Figure 3.12 shows an example of a simple architecture that we used to evaluate and identify the 'data' components/connections. In this example, we studied the data passed through the connections/components and classified these data based on the consequences if lost. Hence, we evaluated the components/connections and identified

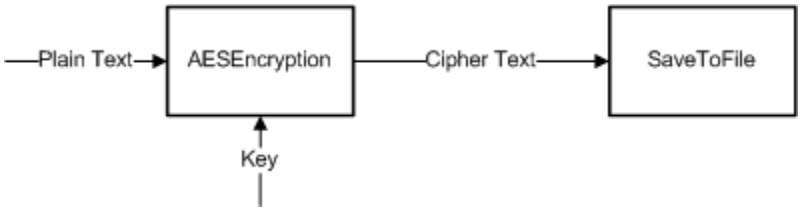

Figure 3.12. Simple Architecture

the critical ones. Since the plain text and encryption key, which is considered critical data, are passed in clear-text format through the input links shown in Figure 3.12, we consider them 'data' connections. In addition, since the AESEncryption component manipulates and processes clear text and keys in clear-text format. Therefore, the AESEncryption is considered 'data' component. On the other hand, the output of the AESEncryption is in cipher-text format; thus, the connection is considered 'control' connection. The same applies to the next SaveToFile component due to the fact that it performs the write to desk on cipher-text format. Consequently, the SaveToFile component is considered 'control' component. Figure 3.13 shows the architecture diagram after conducting the security evaluation and identifying the 'data' and 'control' components/connections.

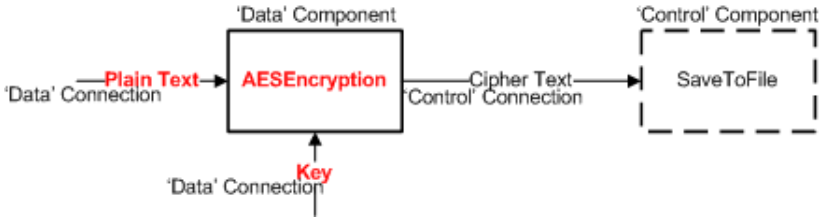

Figure 3.13. The Simple Architecture After identifying the Critical Components

After evaluating the architecture's components/connections to identify the critical ones as shown in Figure 3.13. We applied the NVASA framework to these critical

components. And by implementing the vFork, which acts as the Routing Layer, and the vJoin, which acts as the Decision Layer. We have managed to minimize the overhead associated with the NVASA framework compared to implementing the NVASA framework to the entire system. Figure 3.14 shows the new architecture after applying the NVASA framework to the critical components.

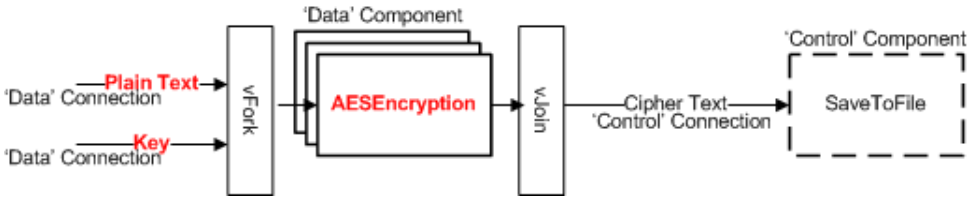

Figure 3.14. The Simple Architecture After Applying the NVASA Framework to the Identified Critical Components

3.3.2. Source-to-source Language translator

Another way to achieve limited diversity is by using source-to-source translators [100]. After producing the initial single version and verifying the design. It can be used to generate multiple versions using source-to-source translators. For example, a Java program can be translated to C/C++ using Toba [24] or to Python using java2python [25], many other translators (commercial and free) can be used in conjunction with other languages [10, 18, 33].

Source-to-source translators will definitely reduce the overhead introduced by the NVASA framework, but will lose some diversity because translating the initial version will produce multiple versions with the same algorithm, data structure and design. Therefore, source-to-source translators will be less effective than producing N-Versions by N groups of developers.

3.3.3. Cross Platform Compilers

One way to achieve N-Version diversity is to compile the source code via different off-the-shelf compilers [100]. This approach was proven to produce diverse machine-code versions that can tolerate compiler introduced errors. However, this approach is less effective than the traditional N-Version programming because it will use the same design, data-structure, and algorithm. Therefore, losing the diversity produced by the N groups. Figure 3.15 shows how to achieve this compiler diversity through different off-the-shelf compilers.

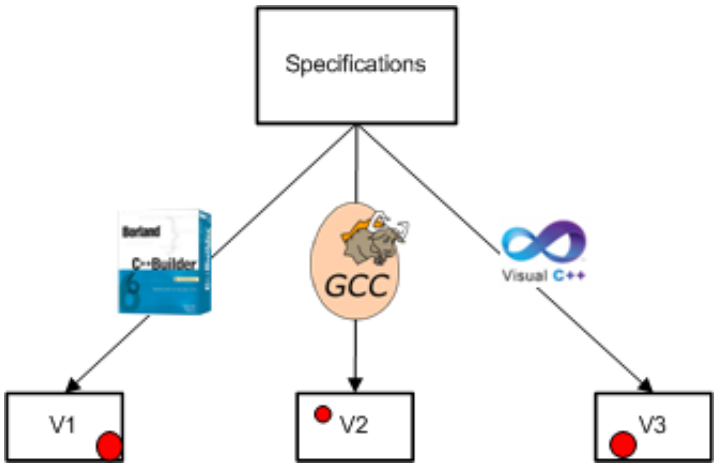

Figure 3.15. Cross Platform Diversity [100]

3.3.4. Study of how the Compartmentalized NVASA Framework would Detect and Prevent the Top Application Vulnerabilities

Here, we examined how the NVASA framework would detect and prevent each attack mentioned in the previous Chapter (For more detail on each attack, please refer to Section 2.2.7). We assume that the operating system OS hosting the application is hardened and sandboxed and that any connection to/from the application is done

through secure channels and through specific port/machine to prevent any side attacks targeting the OS or firewall.

3.3.4.1. SQL Injection Attack

By applying the NVASA framework to the critical parts, we would N-Version the first-line of functions that receives the user input into three or more versions. Each version would filter the input by escaping the SQL characters. Each version would generate the query separately. Some of the new languages like Java can use parameterized queries [19]. Parameterized query is a method where the query is defined by the developer beforehand, the dynamic values are supplied later at execution time. This method would escape SQL meta-characters automatically [32]. .NET also provides parameterized query methods. Each language library can be bypassed uniquely [32]. Thus, the language, and filtration diversity, and the unanimous decision layer algorithm would detect the malformed query. As a result, prevent any malicious query from reaching the database.

In addition, for ultimate security, the database could be N-Versioned as well. Therefore, capturing any malicious query that bypasses the N-Version first-line function. In this case running the query against the N different databases will exploit one or more of the databases due to the uniqueness of the database type discussed earlier in table 3.1. As a result, at least one database would flag the query as faulty. Hence, giving the unanimous decision layer the capability to detect and revert the query for all N databases preventing the exploit from taking place.

3.3.4.2. OS Command Injection Attack

Applying the NVASA framework to the functions receiving the input would capture the malicious OS command input through the N-Versions and decision layer.

This would be successful because some attacks (Commands) are specific to certain OS and can be preventable through OS diversity. In addition, some languages enable developers to set boundaries to restrict any commands from executing beyond certain privileges. For example, the following Java command enables this restriction: java.io.FilePermission;. Therefore, the NVASA framework diversity in language, and hosting OS would enable the decision layer to detect and prevent any OS commands from executing.

From the previous code example mentioned in the privous Chapter:

```
int main(char* argc, char** argv) {
                char cmd[CMD_MAX] = "/usr/bin/cat ";
                strcat(cmd, argv[1]);
                system(cmd);
        }
```

By applying the NVASA framework to the section before the system() command, we would have three or more versions written in other languages such as Java, C# and Ada, etc. Other languages access file differently; in addition, C# runs on Windows based systems only. Therefore, the decision layer will receive the content of the file from three or more functions before applying the system command. The system command would then display the consensus on the console. The C function being exploited will pass the injected command while the other versions would capture the command or fail differently. Thus, enabling the unanimous decision layer to detect the attack and prevent it from executing.

3.3.4.3. Classic Buffer Overflow Attack

Applying the NVASA framework on the first line of functions would detect and prevent a BOF within these functions. Now what if the BOF vulnerability was in a second or third line of functions? In this case if the buffer within the second or further function was smaller than the buffer in the first line of function, then a BOF

attack is possible.

To prevent this BOF attack from taking place after the N-Versioned first-line of functions, we restrict that any buffer within further functions within the path that performs any kind of copying, should be equal to, or greater than the buffer in the first-line function. This could be detected through automated parsing tools. Thus, guaranteeing that the first-line check performed, apply to further functions as well, preventing any BOF attack from taking place within the application.

3.3.4.4. Upload of Dangerous File Types Attack

By applying the NVASA framework to the first-line of functions, it guarantees a sanitized input before any execution takes place. Each version would apply a unique restriction scheme, for example one version can perform a byte sequence check, while another version could run a virus scan. The attacker can craft the input file to bypass a specific version scheme, fortunately the rest would perform a different scheme and at least one version would capture the attempt. This would enable the unanimous decision algorithm to detect and prevent the attack from propagating through the system.

For ultimate security, we can apply the NVASA framework to the anti-virus scanner responsible for checking all uploaded files. This would guarantee detecting any malicious files. Corresponding work in virus analysis through multiple anti-virus scanners has been done. One example is VirusTotal a free online service [44]. In this case if one, or a certain threshold of the N anti-virus scanners reveal the file as malicious, the unanimous decision layer would detect the threat and block the file from propagating through the system.

3.3.4.5. Cross Site Scripting (XSS) Attack

In XSS attacks, we are mainly concerned with the server-side application. Therefore, by applying the NVASA framework to the first-line of functions that receives the malicious script, all N-Versions will take the input and escape untrusted data and perform a whitelist validation on the external input [112]. As a result, producing sanitized input, which will enable the decision algorithm to block the input from propagating through the system if it contained any scripts. In this case, an advanced attacker can succeed in bypassing one version's sanitization/filtration techniques with a specific crafted input, but the rest will expose the malicious input or will remove the untrusted data. The decision algorithm here should come to a unanimous decision. Thus, blocking the scripts from propagating through the system.

3.4. Chapter Conclusion

In this Chapter, we have introduced the usage of the N-Version programming methodology to automate security through our proposed NVASA Framework. We explained in detail the building blocks of the NVASA framework and how they interconnect. The first layer is the N-Version routing layer, which acts as the interface of the application. The routing layer receives each request and routs it to the versions within the N-Version environment layer to be executed. The second layer is the N-Version environment layer where the N-Version resides. The third layer is the N-Version decision layer. The decision layer core function is to run a consensus algorithm to look for a majority output from all N-Versions. For ultimate security, a unanimous output is sought instead of a majority output. The fourth and final layer is the N-Version backend layer. The backend layer contains the application data. It could be in any sort, a files, database, etc.

Following that we introduced reduction schemes to minimize the overhead associated with the implementation of the NVASA framework. We introduced com-

partmentalizing the application into two or more components. Moreover, applying the NVASA framework to the critical components exclusively instead of the entire application. This could be achieved through two methods, 1) automated static analysis through code parsers, 2) or functionality knowledge evaluation done by security architecture experts. We also discussed other overhead-reduction schemes such as cross-platform compilers and source-to-source language translators to achieve limited diversity.

Finally, we investigated some of the top known practical application vulnerabilities that threaten 'application security' and exhibited how our compartmentalized NVASA framework would, nonetheless, protect the core application from successful attacks while reducing the overhead associated.

In the next Chapter, we introduce the usage of the NVASA Framework in Cloud Computing to consolidate its resiliency, and benefit from the cloud computing attributes.

Chapter 4

NVASA FRAMEWORK FOR CLOUD COMPUTING APPLICATIONS

4.1. Chapter Introduction

Due to some limitations in building the NVASA Framework on static infrastructure, we decided to investigate running the NVASA Framework on a virtualized environment (Cloud Computing Environment) to take advantage of the cloud characteristics and avoid the limitations within static infrastructure.

One of the limitations discovered when building the NVASA Framework in static infrastructure from a security point of view are attacks targeting the state of the exploited version in the N-Version Environment Layer. For example, if an attacker could exploit one of the applications (version) in the N-Version Environment Layer, theoretically, he would be able to open a port on the exploited machine and invoke a shell to listen on that specified port. Consequently, connecting to the port later and gaining unauthorized access to the exploited machine. We can see that despite the correct output the framework produced, the attack have exploited the state of the host machine through our application. We will explain how to mitigate these attacks later in this Chapter.

4.2. Cloud Computing Paradigm

Cloud Computing is still an evolving paradigm. It is defined and observed differently by various institutions and providers. Some of these institutes view cloud computing as renting services. For example, renting virtual and web servers, storage

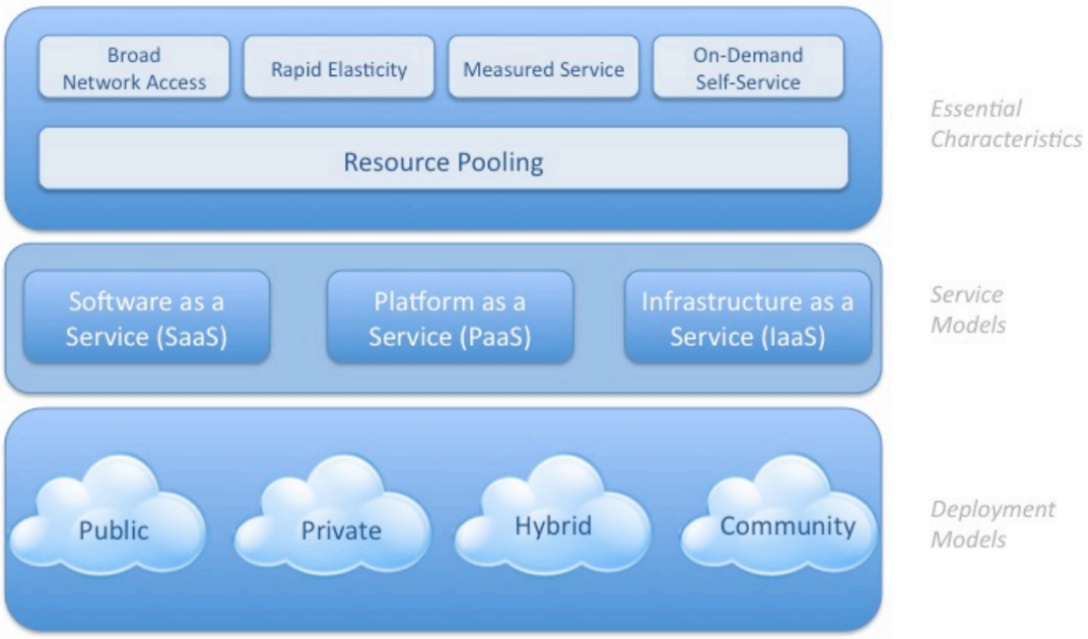

Figure 4.1. Cloud Services and Deployment Models [56]

spaces, and/or processing power. Others view cloud computing as infrastructure, network, and/or computer outsourcing. Some believe that cloud computing is the next generation of networks and service providing. In regards of this praxis, we will view cloud computing as providing services over an untrusted environment and connections over the internet.

In cloud computing there are three service models and four deployment models offered by cloud providers today. As shown in Figure 4.1 [56]; these services could be Infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Software as a Service (SaaS) [81]. And based on the infrastructure of the cloud it could take one of four cloud deployment models; public, private, hybrid, or community model.

The Cloud computing paradigm offers many benefits, including [81, 108, 56]: 1) reducing overall system cost, 2) reducing run time and response time, and 3) increasing

automation. Reducing overall cost because networks, infrastructures, and applications are rented as a service where no purchase is necessary. As a result, eliminating the capital investment. Furthermore, these cloud services are paid incrementally. thus, eliminating or reducing hardware idle time.

Reducing run time and response time is achieved by optimizing the CPU usage by farming workers, virtual servers, and refactoring applications as the demand increases.

Systems are highly automated using virtual servers and a central API controlling the infrastructure; automation is easy and more effective. For example, applying patches can be performed automatically. Therefore, there is no need to carry updates separately for each server. Automation reduces efforts and expenses significantly in cloud computing.

However, with every newborn paradigm come new challenges unique to its abilities and characteristics [66]. In cloud computing, security is a huge challenge where confidentiality, integrity, privacy, and resiliency of the provider's application programming interface (API) and virtual environment are among the biggest issues facing its progress and success.

4.3. Issues with Security in Cloud Computing

Security is a major aspect and concern to customers when choosing a cloud computing provider. Consequently, security will affect the customer's buying decision based on the provider's reputation for security level of their cloud's API. In a recent survey, 2100 top IT security managers were asked if security would get harder, stay the same, or get easier in the emerging cloud computing paradigm [85]. One-third said it will get easier, one-third said it will stay the same while one-third said it will get harder. This survey shows the lack of a consensus and shows how even security specialists are divided in their opinions when it comes to security in cloud computing.

Cloud computing poses challenging security risks because confidential data is no longer confined to local and controlled servers and networks; rather, data reside on public/hybrid clouds on virtual servers on virtual networks. As a result, private and sensitive data protection becomes very important, even monitoring the data and checking the effectiveness of the protection schemes can be difficult to measure. Furthermore, shared resources is an essential characteristic of cloud computing; therefore, isolation of storage spaces, memory, and routing schemes, etc. are very challenging and failure could result in costly consequences.

In addition, cloud customers manage interfaces of the cloud through their own applications, which in most cases are accessible through the Internet. Hence, this issue increases the probability of exploiting these applications through their vulnerabilities. Consequently, risking illegal access to other cloud customer's private data. And since security is only as strong as its weakest link, the provider's cloud security will be potentially measured by his weakest customer's application security. Therefore, a vulnerable customer application could bring the cloud's security level down and would threaten the rest of the provider's customers if exploited.

For cloud computing to reach its full potential as service architecture or technology of choice, it must offer concrete information and network security.

4.4. Related Work in Cloud Computing

4.4.1. Virtualization Characteristics

Cloud computing virtualization offers techniques and methods to automate and improve system security, availability, and reliability [108]. Some of the methods to improve security are: first, 'Sandboxing' the virtual machines to isolate risky and critical virtual machines and their applications by restricting their access to memory,

network, and resources to reduce the overall risk. Thus, improving the overall system security. This is achieved in real life in cloud computing by running risky applications on separate virtual servers and restricting their access to hardware, storage space, and resources to minimize or limit the consequences if an attack took place.

Second, virtualization provides an instant safe restore point for compromised virtual machines to an initial clean state. This is achieved by monitoring virtual servers for unstable behavior or failure. As a result, if a compromised server's behavior was detected; it is restored to a safe clean state. Thus, decreasing or eliminating the risk of potential malware, viruses, or attacks targeting the state of the machine. Therefore, increasing the overall system security without negatively affecting the availability of the service.

Third, the ability to automate virtual machine security patching, is essential for cloud computing security, the automation reduces the effort of applying security patches and decrease the risk of exploitation when a vulnerability is discovered; meanwhile, keeping the servers up-to-date. In virtualization this can be done by updating and applying newly released patches to the initial clean copy stored by the cloud's API. Therefore, cloning the rest of the running machines from the updated initial clean copy and shutting the un-updated ones down. This greatly reduces maintenance, and down-time; thus, increasing the service's reliability, availability, run-time, and security. Methods introduced in our approach would take advantage of these characteristics offered by cloud computing and virtualization to strengthen the NVASA framework implementation and to make our application more resilient and capable of withstanding cyber-attacks without losing sensitive data. Hence, improving the overall application security.

Other methods like forensics and virtual honeypots can be used in cloud computing to enhance and strengthen security. In addition, virtual intrusion-detection systems

could be more effective and cheaper to implement in cloud computing. These methods are used to increase and automate security in cloud computing while reducing the overall cost and overhead.

4.4.2. Related Work in Cloud Computing Security

While cloud virtualized architectures vary, typical elements of such architecture is presented in Figure 4.2 A virtualized, controlled interface isolates the first tier of application servers from the external customer sites or external clients. The application tier consists of potentially load balanced web servers, which respond to web page requests from external, untrusted sources. The application tier interfaces with a virtualized data tier backend repository such as a relational database. In Figure 4.2 backups from the virtualized database servers are stored in a secure permanent data store. Cloud security controls that are typically supplied by the infrastructure provided are depicted in Figure 4.3 The protected boundaries/controls are secured to protect infrastructure and generic services and data and to ensure compliance with legal and industry-specific certification requirements such as the Sarbanes-Oxley Act and the Health Insurance Portability and Accountability Act (HIPAA).

While cloud security architectures are carefully engineered to appropriately address risks to the cloud and customer organizations, unfortunately customer deployed application software security cannot be addressed. Furthermore, software applications pose the highest security risks in a customer cloud deployment. The Open Web Application Security Project (OWASP) reports the top ten web application security vulnerabilities citeowasp10. Of the top ten risks reported, all of these potentially affect cloud customers. This reinforces the observation made in [73] that incorrect or missing input validation causes most vulnerabilities in web applications. Since complicated systems typically possess unknown application security flaws, a solution

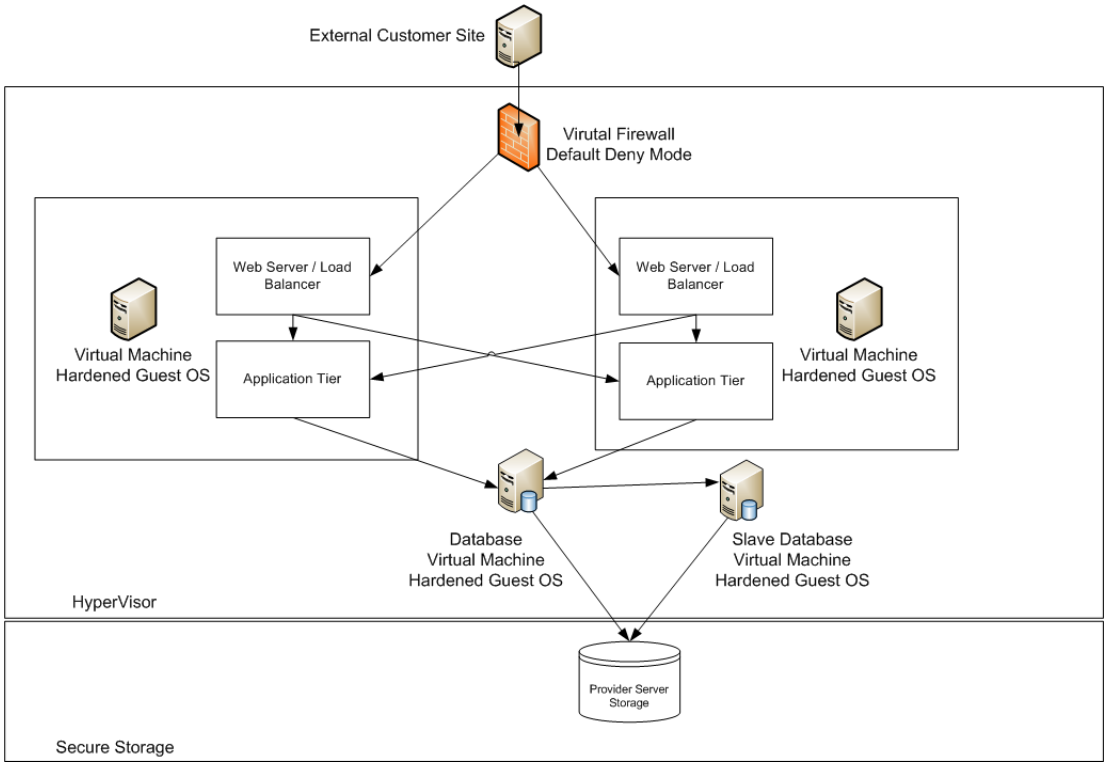

Figure 4.2. Typical Small System Cloud Architecture

that provides an increased assurance level for critical data is often cost effective.

IBM Cloud IBM is one of the leading companies to implement cloud computing. They emphasize heavily on cloud security and invest tremendous amount of time, effort, and money on researching security schemes in cloud computing. IBM divides their responsibility of security into three levels based on the service they provide to their customers. First, customers using (SaaS) are minimally involved in the security schemes and procedures; this is because most of the control and management are done through IBM. Second, customers using (PaaS) are more involved in security and hold some control and responsibility; this is because the platforms are usually implemented and mostly controlled by the customer. Therefore, removing some of

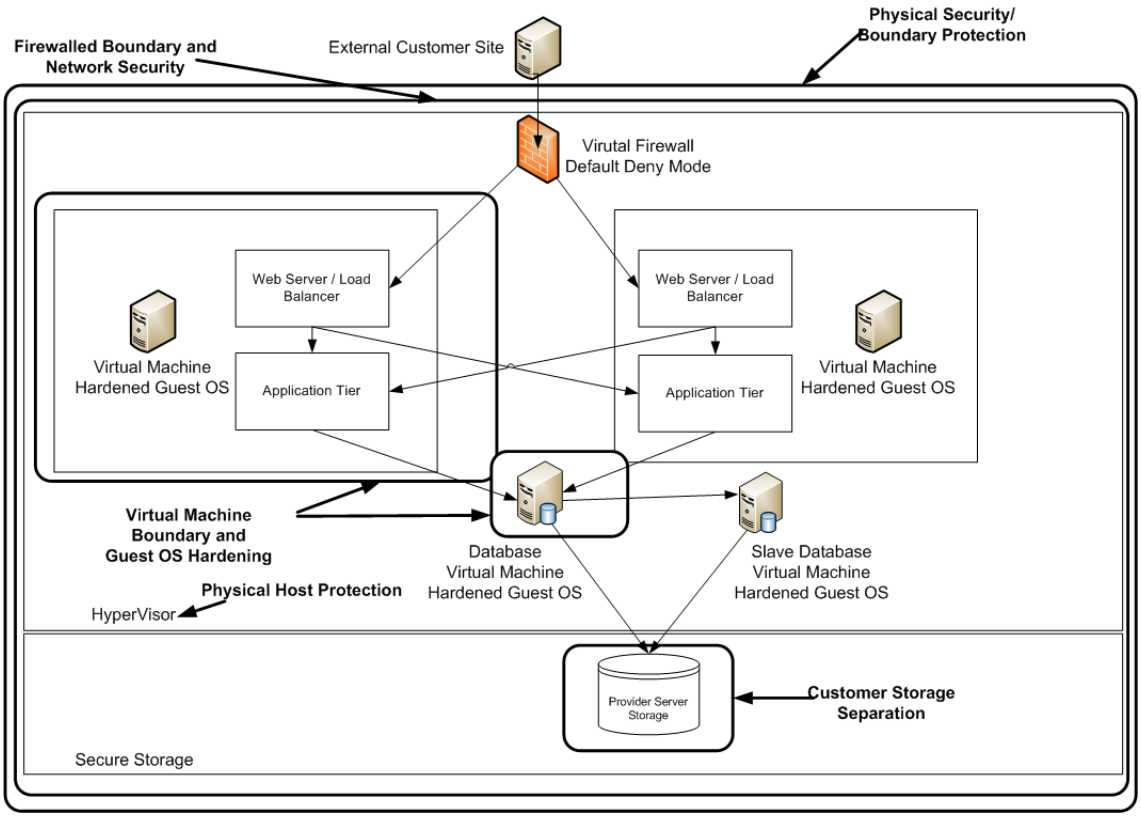

Figure 4.3. Virtualized Cloud Infrastructure Security Boundaries

the control off IBM. Finally, customers using (IaaS) are very involved in security and hold most of the control and responsibility over their infrastructure.

IBM cloud computing security consists of two main layers as shown in Figure 4.4 [105]: a service-oriented architecture (SOA) [65] layer that resides above the new secure virtualized runtime layer. The SOA in general terms is a set of design ideology used to enable the integration of services to be used by multiple businesses. The IBM SOA security layer enforces a set of rules and regulations to 1) authenticate and authorize, 2) Audit compliance, and 3) Isolate subscriber's domains and integrate their existent security infrastructure with theirs. The secure virtualized runtime layer

is a virtualized system beneath the SOA layer that runs the processes that access the data storage. This layer operates on virtual machine images instead of operating on individual applications. Hence, the run time layer provide security services around the virtual machines via anti-virus, host intrusion detection, and introspection.

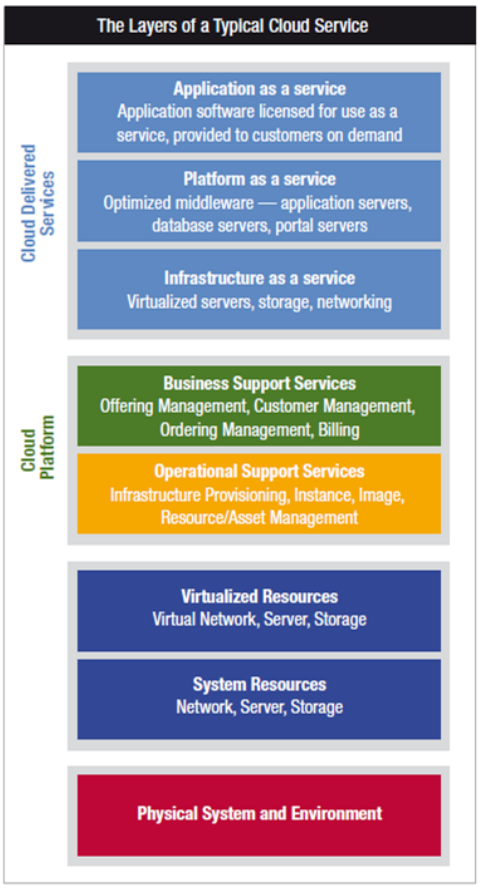

Figure 4.4. Layers of IBM Cloud Service

AmazonEC2 Amazon is also one of the leading companies in cloud computing. They entered the cloud business in August of 2006. Amazon's EC2 system security is structured over multiple levels [103]. First, at the host operating system level the administrator is required to pass multi-factor authentication schemes to gain access

and control over the cloud. Second, the VM guest operating system (OS) is pre-configured and hardened, and controlled fully by the customers. Customers have administrator privilege over the guest OS to control the accounts, applications, and services running. Amazon claims that they have no access rights to the guest OS belonging to customers. Third, firewalls provide protection on a port level. Traffic may be restricted by source IP, protocol, and service port. Forth, API calls exist to enable the customer to terminate instances, re-configure firewall parameters, or perform other functions. These calls are all signed by an X.509 certificate or the customer's key. Amazon recommends encrypting the data using SSL but does not mandate it.

On a lower-level Amazon isolates the virtual machine instances using the Xen hypervisor isolating customers from direct access to raw drivers and instead accessing virtualized disks. In addition, Amazon EC2 resets every data block after the customer's application is done with. Thus, preventing any un-intentional exposure of the information to other customers or cyber-attackers. Amazon also recommends that the customer takes the appropriate actions to secure and protect their information by encrypting their file system or their virtualized images.

Amazon emphasizes that bugs, and vulnerabilities will occur over time. Therefore, they strongly advise customers to ensure strict guidelines are used to test their applications and services and to apply patch and configuration management as needed. Furthermore, they emphasize the adoption of good coding practices to minimize the introduction of errors during the development phases which will introduce vulnerabilities [104].

Amazon and IBM invest tremendous time, effort and money on cloud computing security. They are aware of the impact it has on their survival and growth in the industry. However, with their considerable effort, there is still high risk of at-

tacks targeting the application layer. This could lead to unauthorized disclosure of their customer's sensitive and private data. Statistics show that tens of severe vulnerabilities are discovered every year within operating systems run by Amazon and IBM clouds. In addition, statistical reports also show that many online applications/services are severely vulnerable due to un-detected application vulnerabilities [111]. Therefore, despite their tremendous effort, there is a risk of exploiting these vulnerable VMs or the applications/services running on them to gain unauthorized access to the cloud or portion of it. Implementing the NVASA framework in cloud computing complements existing security implementations to enhance application security; and therefore, enhance the overall cloud's security.

4.5. Implementing NVASA Framework in Cloud Computing

Cloud computing structure is divided into several layers where each layer performs certain tasks. These layers combine to build the concept of cloud computing. Figure 4.5 shows the sequence of the layers in a cloud computing environment. From a top-bottom point of view, the first layer is the application and services layer which runs on the isolated VM instances. The second layer is the NVASA framework layer. The NVASA framework layer is a security layer that resides beneath the applications/services layer. The NVASA framework layer protects the services/applications layer. Therefore, protecting the virtual machines and the data saved on them. The NVASA framework layer protects the applications and services from being exploited by certain attacks targeting existent vulnerabilities. The third layer is the actual VM and storage layer where all the processes are initiated, and data are virtually stored. The fourth layer is the virtual platform layer where all the VMs run on one or more virtual platforms. The fifth layer is the virtual infrastructure layer that holds the virtual platforms. The sixth layer is the Hypervisor layer which is accountable

for isolating the virtual layer from the physical layer. In addition, the hypervisor is responsible for controlling the shared hardware among the virtual environment. This is achieved by separating the privilege modes into 3 or more modes and restricting their access based on their prioritization. The seventh layer is the physical cluster where all the hardware is placed to run the virtual environment.

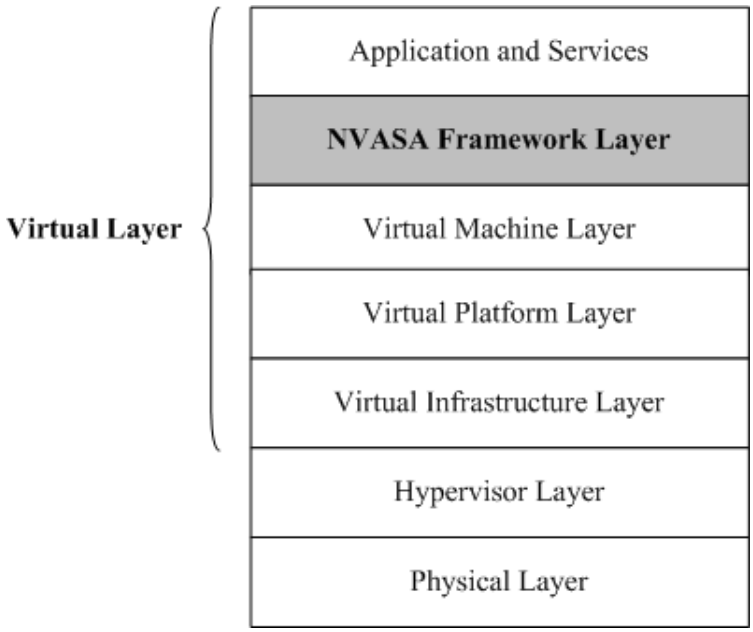

Figure 4.5. NVASA Framework Layer within Cloud Computing Structure

The NVASA framework enhances the overall system security by improving resiliency through redundant and independent service implementations. This architecture protects and strengthens the application/service layer by helping to minimize the risk of exploiting existent vulnerabilities. Thus, protecting sensitive data in the VM and Storage layer from being stolen or damaged. To implement the 4-layer NVASA framework shown in Figure 3.2 into a virtualized environment, we compartmentalized each layer in a separate VM as shown in Figure 4.6. Similarly, the Virtual N-Versions within the Virtual N-Version Environment Layer were separated and executed on

separate VMs as well.

4.5.1. Virtualized NVASA Framework

Virtual N-Version Routing Layer

First, the Virtual N-Version Routing Layer, is the external interface of the system and is responsible for routing requests to each component of the Virtual N-Version Environment Layer for execution. The Virtual N-Version Routing Layer should be built using a dynamic language and application programming interface (API) to enable communication with heterogeneous versions written by various languages and APIs that are virtualized on different operating systems. The VM hosting the Virtual N-Version Routing Layer should be 'Sandboxed' by disabling all unnecessary features and enabling only the routing of requests to the Virtual N-Version Environment Layer. The Virtual N-Version Routing Layer is also responsible for replying back to requesters with the consensus output passed by the Virtual N-Version Decision Layer. Finally, the Routing Layer may be configured to log "un-trusted" inputs received from un-trusted sources. This input may be correlated with N-Version output exceptions logged by the Virtual N-Version Decision Layer to aid in detection and removal of potential vulnerabilities and version implementation errors, hence improving the quality of the versions.

Virtual N-Version Environment Layer

The second layer is the Virtual N-Version Environment Layer consisting of the Virtual Versions; each Virtual Version should be built using the rules and regulations defined by the N-Version programming methodology mentioned privously in Chapter 3 [58, 82]. Each Virtual Version should run on a separate VM to take advantage of the first characteristic of cloud computing 'Sandboxing'. 'Sandboxing' here empowers the cloud API to restrict and limit the hosting operating system of the Virtual Version

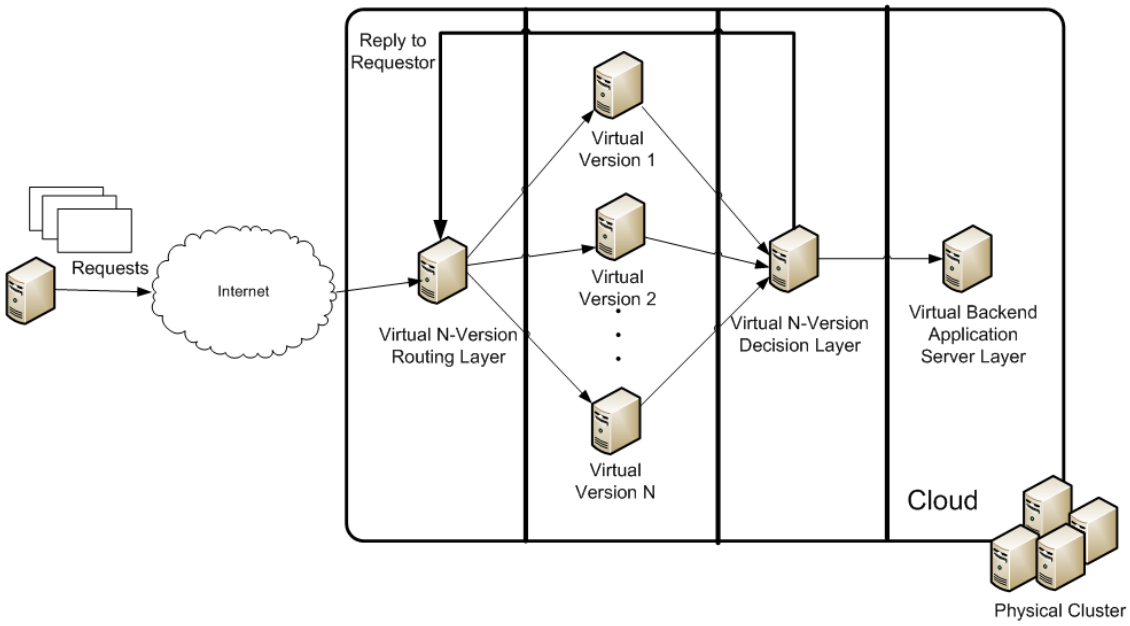

Figure 4.6. Implementing NVASA Framework in Cloud Computing

from accessing unnecessary storage and network resources. This minimizes the risk of exploited Virtual Versions from granting illegal access to the attacker.

Virtual N-Version Decision Layer

The third layer is the Virtual N-Version Decision Layer. This layer holds the intelligence of the NVASA framework. The Virtual N-Version Decision Layer can detect known or unknown attacks by utilizing a consensus algorithm to evaluate all Virtual Versions' output, comparing each output of the Virtual Versions. As a result, a conflicting versions' output is flagged as a compromised version, and will be removed from the pool of independent versions. The algorithm continues until $([N/2] + 1)$ Virtual Versions reaches a consensus output. In addition, we require the Virtual Decision Layer to restore a compromised virtual machine to a safe point. This feature enables the NVASA framework to reset the state of the Virtual Version's machine to a clean and un-tampered state thwarting a potential compromise at the

application level of the system.

The Decision Layer also maintains information regarding the rate that each of the virtual machines is compromised. This information helps defend against Denial of Service (DoS) attacks against an NVASA architecture that would potentially utilize repeated attempts to exploit the same vulnerability of a specific version and initiate repeated virtual machine restorations, hence created a denial of service. If a DoS attack occurs, the Decision Layer would simply stop the vulnerable version and continue processing utilizing a reduced set of versions. The affected version could be patched and brought back on line without significant loss of functionality.

Virtual Backend Application Server Layer

The fourth layer is the Virtual Backend Application Server Layer, where the system data repository is maintained. The Virtual Backend Application Layer is protected by the decision algorithm in the Virtual N-Version Decision Layer where outputs from the Virtual Versions are never directly applied. Instead, a consensus output is applied with protected, secure communication between the Decision Layer and the Backend Application Layer, eliminating exploited Virtual Versions from damaging, tampering, or stealing sensitive and private data stored within the Virtual Backend Application Layer.

4.5.2. N-Version Management and Auditing

The NVASA framework provides layered security services, including monitoring and control, auditing and logging, host intrusion detection and system heartbeats. Monitoring and control are provided in a separate virtual machine running system software such as Nagios [27]. This provides an external management view of the system during mission execution. In addition, the NVASA framework utilizes a separate virtual machine for audit collection and log file reduction. Not only does this aid in

the forensic analysis of the operation of the N-Version Environment Layer, this provides valuable information that is helpful in hardening the implementation in each specific version.

At system startup and whenever a specific N-Version is restored to a safe point, host-intrusion software validates all security-related configurations and the implementation executables to ensure that the system was restored to the correct state and has not been started in a compromised state. Any violations result in logging and alarm events.

Additional monitoring of the NVASA framework state and health includes "Heartbeat" messages between the N-Versions and the Virtual N-Version Decision Layer. This feature is specified as follows; the versions will send a "Heartbeat" message within a specified interval to the Virtual N-Version Decision Layer thereby notifying the Decision Layer that the version is alive. If a version is unresponsive, the Virtual N-Version Decision Layer would mark the version as unresponsive and restore the corresponding VM.

4.5.3. Strict Privilege-Mode of NVASA framework Layers

In cloud computing environments, privilege-mode is essential to restrict access within a hardware cluster to prevent attacks at the processor level. For example, Amazon EC2 provides 0-3 privilege-modes to restrict access to the processor. Therefore, having an authorization scheme to minimize risk. Hence, the NVASA framework virtual machines should be separated by different privilege-modes depending on their role and risk of exploitation. The Virtual N-Version Routing Layer virtual machine should be given the minimal privilege since it is the interface of the framework. In addition, the Virtual N-Version Routing Layer main task is receiving requests and routing them to the N-Versions in the Virtual N-Version Environment Layer.

Moreover, the Virtual N-Version Environment Layer, which contains the virtual N-Versions, should be given the minimal privilege as well since some versions might be vulnerable. Finally, the Virtual N-Version Decision Layer is more resilient than the rest of the layers because of the consensus algorithm. Therefore, needs more privilege to access, delete, add, read, and create data in the Virtual Backend Application Server Layer.

4.6. Chapter Conclusion

In this Chapter, we introduced the cloud Computing paradigm and discussed in detail the characteristics that make cloud computing exceptional and unique, some of which are virtulization and elasticity. The NVASA framework resiliency benefits from the cloud virtulization through sandboxing the N-Versions and layers into separate VMs, and virtual sub-network. Therefore, eliminating or minimizing the consequences of any successful exploit. In addition, the NVASA framework can mitigate side attacks targeting the OS by reverting the VM to a safe state whenever a discrepancy or unexpected behavior is detected. This is accomplished without any difficulty using automated cloud APIs that in addition, would eliminate any down time.

In the next Chapter, we conduct three separate experiments to validate the enhancement of application security through the development of the NVASA framework.

Chapter 5

EXPERIMENTAL RESULTS

5.1. Chapter Introduction

Three experiments were conducted in order to validate that the NVASA architecture does provide an enhanced security posture in distributed environments, as well as cloud-based web service architecture. In this chapter, we will explain the architecture implemented for each experiment, and the results achieved.

5.2. Experiment One: Simple Text Input Implementation

For the first experiment, the Backend Application Server Layer was not implemented. In order to understand the overall scope of the experiment, a use case and sub use case is provided to illustrate the system concept of operations.

5.2.1. Experiment Use Cases

The use case selected to provide the context for our experiment is a Secure Enterprise Access to a Cloud Web Service. The secure cloud web service must be able to accept an un-trusted web service request, route the request through the virtual N-Versions, compare each version's output and provide back-end processing of valid service requests or take appropriate action on a compromised version. The use case is presented in Use Case 1 and 2. Web service requests are submitted to a NVASA Protected Service (NPS) and a response is returned without the client service aware of the internal security mechanisms of the NPS. If the service is threatened and an

attack successfully compromises a virtual service, a correctly implemented Decision Layer would take action to reset the virtual service. Notification of the threat by the Decision Layer object to the Routing Layer object would initiate a Service Abuse Policy by the Routing Layer object.

Secure Enterprise Access to a Cloud Web Service	
Primary Actor:	Enterprise or Cloud Web Service
Scope:	NVASA Protected Service
Level:	User Goal
Main	Success
Scenario	1. Service client crafts a service request for the protected service 2. Service request is forwarded to the service endpoint/routing layer 3. The routing layer forwards the service request to each virtual service version 4. Each virtual service computes a service response and forwards to the decision layer 5. The decision layer compares each virtual service response 6. The decision layer forwards the service response to the routing layer endpoint 7. The routing layer returns the service response to the client
Extensions:	1. Decision Layer finds discrepancies in the virtual service responses a. Decision Layer applies a virtual version reset policy

Use Case

Apply a Virtual Version Reset Policy	
Primary Actor:	Decision Layer object
Scope:	NVASA Protected Service
Level:	subfunction
Main	Success
Scenario	1. Decision layer determines the invalid response 2. Decision layer determines the affected virtual service 3. Decision layer requests the service host to reset/rollback the service to a known state 4. Decision layer updates the routing layer object 5. Routing Layer object applies a Service Abuse Policy
Extensions:	

Sub Case

NVASA Framework Prototype Implementation An implementation description for each layer of the NVASA architecture is described below. The imple-

mentation is depicted in Figure 5.1.

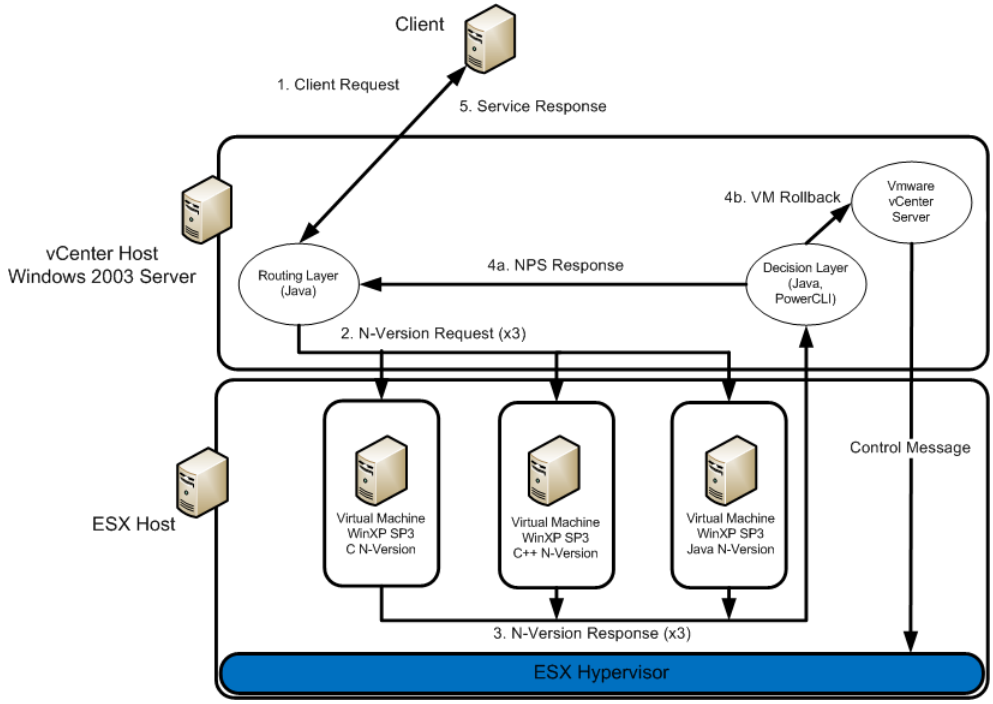

Figure 5.1. Applying NVASA to Object-Oriented Architecture Style

The prototype equipment consisted of three physical machines and three virtual machines (VMs). A single physical host was installed with VMware ESX 4.0 and hosted the VMs utilized. A second physical host was installed with Microsoft Windows 2003 Standard Edition Server and hosted VMware vCenter 4.0 to manage the ESX host and VMs. The processes implementing the NVASA Routing Layer and Decision Layers were also deployed on this machine. A third physical machine running Microsoft Windows XP SP3 was utilized to generate test data and attack the N-Version VMs.

Microsoft Windows XP SP3 was installed on each of the three VMs deployed on the ESX host. Each instance of the NVASA N-Version service was installed separately

in these VMs.

Routing Layer The N-Version Routing Layer was deployed on the Windows 2003 Server and received user data from the external "test" host. The Routing Layer prototype was implemented using Java and duplicated client input to each N-Version implemented in the N-Version Environment Layer. In typical web application architecture, the N-Version Routing implementation would be deployed in the Application Tier.

Decision Layer The N-Version Decision Layer was also deployed on the Windows 2003 Server. The Decision Layer process was implemented using Java and contained the Response Comparator functionality. For this experiment, the Decision Layer was implemented to simply log successful N-Version comparisons in lieu of writing to the Backend Application Server Layer. Upon a failed N-Version comparison, meaning one of the N-Version implementations did not produce results consistent with the other versions. The Decision Layer invoked a PowerCLI script to communicate with the VMware vCenter system and the ESX hypervisor to reset the VM hosting the N-Version to a known, safe restore point using a previously generated VM snapshot.

N-Version Layers For purposes of testing the NVASA architecture, three NVASA virtual services were implemented to accept client service input from the Routing Layer and provide output to the N-Version Decision Layer. The three N-Versions were deployed on three separate Microsoft Windows XP SP3 VMs running on the ESX hypervisor. The three N-Versions were implemented in C, C++, and Java respectively. The C implementation exhibited a string format vulnerability. The C++ version was implemented to contain a buffer over flow vulnerability. The Java version did not have any known vulnerability.

Test Cases/Results To test the NVASA Protected Service, the Main Success Scenario of the Secure Enterprise Access to a Cloud Web Service Use Case was exercised. Since no backend processing was implemented for this experiment, the Decision Layer returned a pass/fail response to the Routing Layer. The system was operated under normal conditions, and the client was unaware of the service protection. The NPS did not affect the normal processing case. Following the happy path testing, the NPS was threatened using a parameter tampering technique designed to exercise a code injection or input format vulnerability in a single service as shown in Table 5.1. Upon failure of the affected N-version virtual service, the Decision Layer correctly applied a virtual version reset policy and restored the virtual service. This action relied on functionality in the vCenter server to rollback the VM to a known restore point. A failure response was then returned to the Routing Layer implementation of the architecture. The NPS was unavailable during restoration of the failed service. It was noted that this was a potential denial-of-service (DoS) attack.

The last test conducted was similar to the initial failure test, but involved exercising a known buffer-over-flow vulnerability. System behavior consistent with the initial failure test was observed, but with a different virtual service. This concludes that the NVASA framework detected and prevented the input attacks from propagating through the virtual environment; therefore, enhancing the overall cloud security.

5.3. Experiment Two: AES Implementation

A second prototype was implemented, and testing conducted to validate the protection provided by leveraging an N-Version service implementation in a distributed or web application architecture. The AES prototype was developed in a private cloud environment. We will discuss the cloud's setup and configuration later in this chapter.

Table 5.1. Test Cases and Results of the Simple Text Input Implementation

Test	Test Case	Test For	C	C++	Java	NVASA
1	%08x%08x	Format String	✗	✓	✓	✓
2	%d%d	Format String	✗	✓	✓	✓
3	34 Byte String	BOF	✓	✗	✓	✓
4	50 Byte String	BOF	✓	✗	✓	✓
5	ssss\\ssssssss	Quotation and Backslash	✓	✓	✓	✓
6	#435435	Special Character	✓	✓	✓	✓
7	NVASA	Normal String	✓	✓	✓	✓

In order to develop a practical prototype that can produce reliable results, we searched for pre-developed AES implementations online to be used as our N-Versions. These N-Versions must be written in different languages and by different developers in order to satisfy the diversity required in the N-Version programming methodology mentioned earlier in Chapter 3.

After much search and testing, we could find three very close implementations of AES. These implementations were similar in structure but written in three different languages and were developed by three different developers or groups of developers. As mentioned in Chapter 3, the development of the NVASA versions, the "version Fork" (vFork), and the "version Join" (vJoin) should be done by following the NVASA Policy given to the developers by the design experts, but in our experimental case, this was different because the three implementations were already developed. Therefore, we had to develop the vFork and vJoin afterwards and had to accommodate all variations between the implementations. This would have been avoided if we followed the sequence by obtaining the NVASA Policy beforehand.

The first version was written in Java by Neal R. Wagner [110]. The second version was written in C# by James McCaffrey [84]. And the third and final version was

written in C/C++ by Niyaz PK [88].

5.3.1. Setting up the Private Cloud Environment

5.3.1.1. Eucalyptus Cloud Architecture

Eucalyptus is an open source private cloud platform [16, 46, 45, 91], it is compatible with Amazon EC2 and S3 services. To deploy Eucalyptus, it requires a Cloud Controller to control the Cloud Cluster. The Cloud Controller also usually acts as the Walrus Storage Service, the Cluster Controller, and the Storage Controller. As the cloud grows and more VMs are required more Cloud Controllers are installed. Figure 5.2 shows our Private Cloud Infrastructure using Eucalyptus. In our setup, we have installed the Walrus Storage Service, the Cluster Controller, and the Storage Controller on the same Cloud Controller. We have installed two Cloud Nodes.

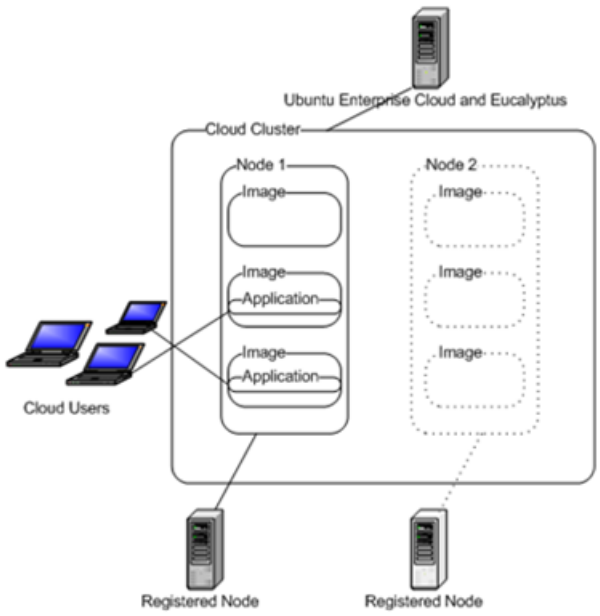

Figure 5.2. Eucalyptus Private Cloud Infrastructure

Our private eucalyptus cloud could handle 24 small VMs. Figure 5.3 shows the cloud capability in terms of the number of VMs. Eucalyptus can provide six different VM sizes. m1.small starts with one CPU and 192 MB of RAM, and goes up to c1.xlarge which requires four CPUs and 2048 MB of RAM.

```
majid@malaikacloud:~$ euca-describe-availability-zones verbose
AVAILABILITYZONE    cluster1                    192.168.0.100
AVAILABILITYZONE    |- vm types                 free / max   cpu   ram  disk
AVAILABILITYZONE    |- m1.small                 0024 / 0024   1    192     2
AVAILABILITYZONE    |- c1.medium                0024 / 0024   1    256     5
AVAILABILITYZONE    |- m1.large                 0012 / 0012   2    512    10
AVAILABILITYZONE    |- m1.xlarge                0012 / 0012   2   1024    20
AVAILABILITYZONE    |- c1.xlarge                0004 / 0004   4   2048    20
```

Figure 5.3. Eucalyptus Private Cloud Capability

At this stage, we downloaded few VM images from the Eucalyptus store, Ubuntu 9.10 (32-bit) and Ubuntu 10.04 (32-bit). There was a software bug in the certificate check process of the Eucalyptus Cloud Controller. We had to modify the Eucalyptus code to accept certificates without performing any checks. This should not be done in a production environment due to the fact that it could be used as a vulnerability to exploit the Cloud Controller.

Eucalyptus does not support Windows by default. Therefore, we had to decide either to exclude any Windows-based platforms; such as, .NET and C# or find another platform that has the flexibility to instantiate Windows, as well as Unix VMs. Since we had been using .NET and C# as one of our implementation versions, we have decided to switch to XEN Cloud platform, which supports Windows and Unix by default.

5.3.1.2. XEN Cloud Architecture

Installing XEN Cloud Platform [50] we have noticed that it is more stable than Eucalyptus, and more flexible since we can use different flavors of Windows and Unix for the VMs. We managed to install the XEN platforms on one of the machines and built the same infrastructure as the Eucalyptus. Figure 5.4 shows the XEN infrastructure and the VMs. We used OpenXenManager [28] on Ubuntu to manage the XEN Cloud Platform. OpenXenManager enabled us to view the VMs, create new VMs, configure the memory usage for each VM, as well as configuring the network interfaces.

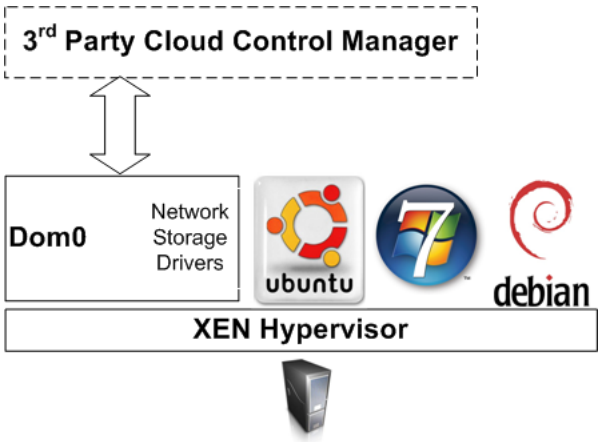

Figure 5.4. XEN Private Cloud Infrastructure

We then installed OpenSSH on all Linux based VMs [47] to be able to easily connect and configure the instances. For Windows 7 instances, we enabled the remote desktop feature to be able to connect and configure Windows instances. Following this step, we installed Apache on the Linux instances [4], and enabled CGI to be able to expose the C/C++ implementation as web services [21]. For the Windows instances we installed IIS 7.5 and enabled ASP.NET to be able to expose the C# .NET implementation as a web service.

5.3.2. Testing the Standalone Implementations

We took each standalone implementation and conducted a security testing by running a benchmark of 100 test cases to detect any existing vulnerabilities. Table 5.2 highlights the most important security findings.

These exploits were successful due to mistakes in the source code. Few vulnerabilities were caused by overflowing the size integer value in the C# and C implementations. This could be avoided by checking the integer size of the user input before continuing the execution. Another set of vulnerabilities were related to file access violations in the Java implementation, once again these exploits could be avoided by checking the validity of the file before attempting to read/write from/to the file. In addition, handling the case if the file does not exist. Another set of vulnerabilities were caused by null input in the Java implementation, once again this could be avoided by checking for null input before continuing the execution. Finally, these implementations accept HEX values. They all failed when inserting a non-HEX value in either the key or plain text. Therefore, causing the program to crash or act unexpectedly.

5.3.3. Building the AES NVASA Framework

As part of the AES framework implementation we exposed the C# and C/C++ versions as web services to be able to integrate them with the NVASA Framework. Each web service would accept a request with a 192-bit key and 128-bit clear text block and reply with the cipher text if the encryption was successful. Since the interface was developed using the Java language, a simple function call was sufficient to connect to the Java version which as well accepts a request with a 192-bit key and a 128-bit clear text block. Figure 5.5 shows the structure of our AES NVASA Framework implementation. We minimized our involvement in writing or modifying any code as much as possible to satisfy the diversity required in the N-Version programming

Table 5.2. Test Cases and Observations of the Standalone AES Implementations

	Vulnerability	**Description**	**Symptoms**
C/C++	No Input Validation in Key Length	Input a char in the key length	Program goes in a never ending loop
	Integer Over Flow (Key Length)	Larger than MAX INT	Integer Over Flows
	Input directly to HEX with no Input Validation	Input a non HEX value for the Plain Text	Program fails giving wrong output
Java	Java Exception not handled with input reading	Provide an empty file	Program crashes
	No Input Validation for plaint text and key	Input a non HEX value	Program uses c0 for any non HEX value
	Exception not handled with File reading	Remove the plain text or key file	Program crashes
C#	No Input Validation for plaint text and key	Input a non HEX value for the Plain Text or key	Program crashes miserably revealing internal variables
	Input validation for the key	Input a larger key than specified	Program crashes miserably revealing internal variables

methodology. Therefore, we used pre-developed implementations that match each version's language to integrate them with the NVASA framework.

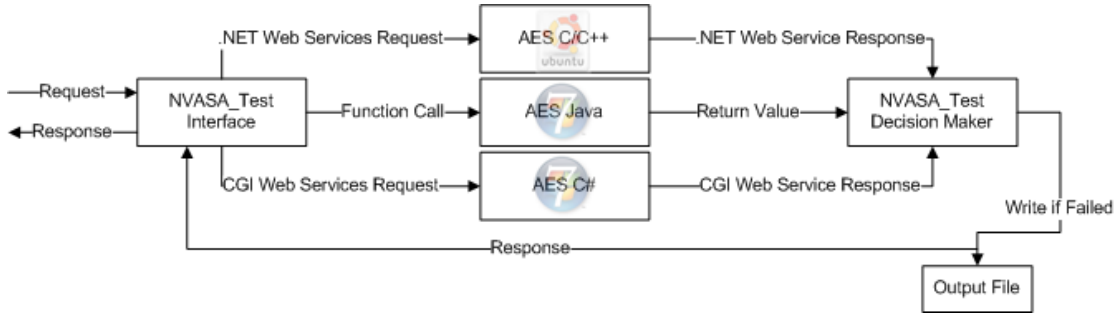

Figure 5.5. NVASA Framework AES Implementation

For this prototype, we wrote a java class to act as the Routing and Decision Layer (Layer 1 and 3 From Figure 3.2). The java class simply accepts the user input then executes all three versions simultaneously. Each version subsequently produces its output, which is then fed to the decision algorithm. The decision algorithm then runs to reach a consensus output. If a consensus is reached, the cipher text is written to a file and sent to the requester. Otherwise, we have the choice of either dropping the request or replying back to the requester with a "Request Denied" message.

To test our NVASA implementation, we developed a benchmark of 100 test cases, a snapshot of the benchmark file is shown in Figure 5.6. Each test case included a key length, key, and a clear text block. We executed each standalone version against each test case first, then executed the same set of test cases against the NVASA AES implementation. Table 5.3 shows a segment of the failed standalone implementations, and the output of each version compared to our NVASA AES implementation.

Looking at table 5.3, we comprehend that in the first two test cases more than one version failed from a total of three. Fortunately, the NVASA framework detected the discrepancy and recovered from such an exploit. This was achieved since each version

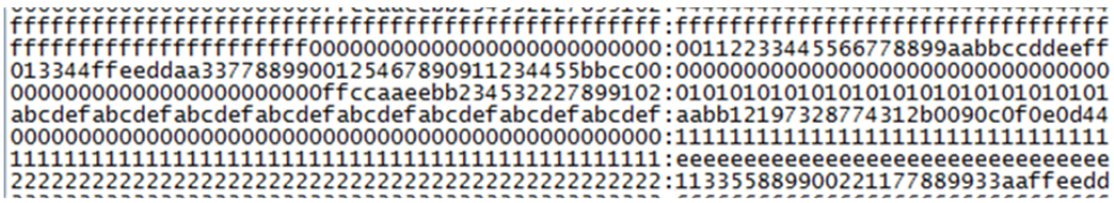

Figure 5.6. Input File Benchmark

produced a different and unique output; therefore, no consensus could be reached at the decision algorithm. Thus, the request was dropped and never propagated through the system. This step is essential because the decision algorithm will detect an attack and prevent it from propagating through the system even if it was successful in exploiting most of the versions.

This NVASA prototype shows how the new framework can improve Security by detecting and preventing known and potential zero-day attacks using the N-Version Programming methodology. The AES Implementation revealed that the diversity of the languages and algorithm used, greatly reduces the probability of having identical vulnerabilities in two or more versions. This was achieved without the need to modify the source code within the versions or install any patches.

5.4. Experiment Three: Moodle Analysis

So far, we have demonstrated through the first two implementations that the NVASA framework has improved the application security of the system. This was achieved even when all versions failed and produced a malicious/wrong output. The NVASA framework was able to recover due to the uniqueness of each wrong/malicious response, which enabled the decision algorithm to detect the attack and prevent it from propagating.

Table 5.3. Test Cases and Results of the AES Implementation

Test Number	Test Case	Test For	C/C++	Java	C#	NVASA
1	73204483711	Integer Over flow in Key Length	✗	✓	✗	✓
2	"zz...." in the plaintext or key field	Input Injection (Non HEX) in the plaintext and/or key field	✗	✗	✗	✓
3	Null Input (Command Line or File)	Exception Handling with plaintext and/or key	✓	✗	✓	✓
4	"Ree" instead of an Integer	Type Injection in the Key length field	✗	✓	✓	✓
5	Normal HEX 192 Key and 128 Key	Normal Input	✓	✓	✓	✓
6	Larger key than specified (200 bits instead of 192 bits)	Input Validation Key Field	✓	✓	✗	✓

In chapter 3 we presented reduction techniques to reduce the overhead introduced through the implementation of the NVASA framework. Therefore, for our third and last experiment, we focused on studying a large implementation called Moodle, an open-source web project to analyze the architecture and code in order to define the critical components. Thus, study the implementation of the NVASA framework to the defined critical components, and the overhead associated with it.

Moodle is an open-source Learning Management System (LMS) project [26]. It is a free web application that enables educators to effectively create an online learning websites and environments. Moodle is written using the PHP language and has the capability of running on Windows or Unix environments and can connect to different types of databases like PostgreSQL, MySQL, Microsoft SQL Server, and Oracle. Moodle is built of 5061 PHP files and 1,060,585 lines of code [26].

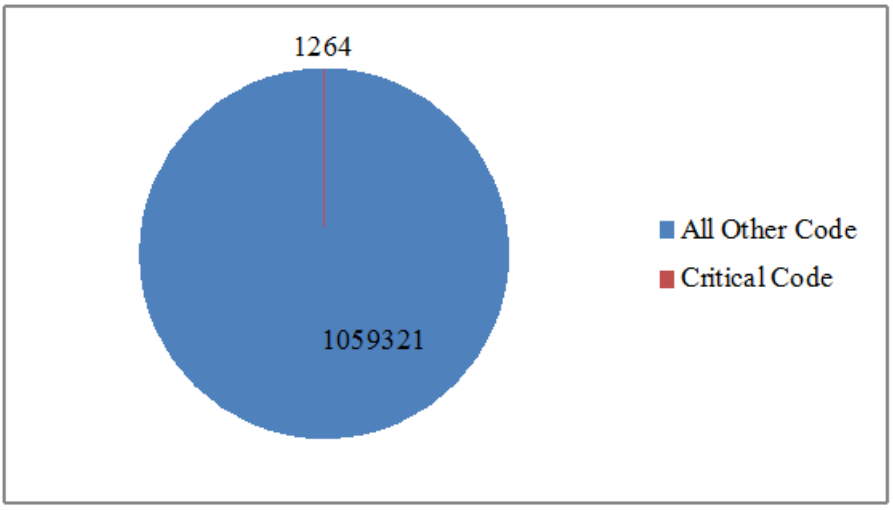

Figure 5.7. Proportion of Critical Code Compared to Total Code

Following the Static Analysis method described in chapter 3 we developed a Java program (Code provided in Appendix A). The Java program takes a folder location,

then crawls into each sub folder parsing through the code of each .php file determining I/O, database queries, and file access code. The program subsequently stores all unique files in a separate file for further analysis.

Figure 5.7 shows the critical lines of code (LOC) compared to the total number of LOC. Clearly, the critical code is a very small subset of the total code, just %0.11. Figure 5.8 shows the different types of critical code. We can see that "*mysql_*" code, which represents SQL queries, is 105 LOC. Further analysis revealed that these function calls resides in 10 PHP files only from 5061 total PHP files.

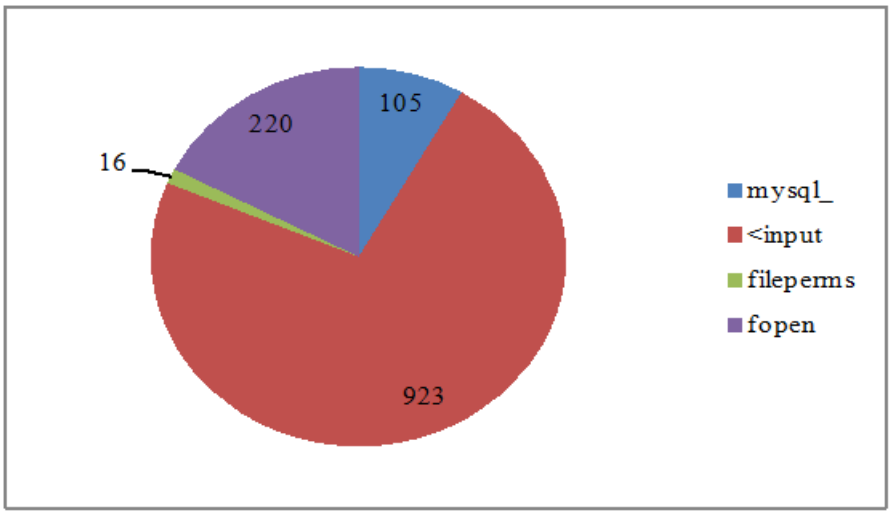

Figure 5.8. Proportion of Critical Code

Another critical code was the "< *input*" commands in PHP. It was found in 923 LOC, further analysis revealed that these command calls resides in 200 PHP files from 5061 total PHP files. Finally, we found 220 LOC of "*fopen*" command, further analysis revealed that these commands reside in 124 PHP files. We took a sample of 20 PHP files with 68 LOC with the "< *input*" commands. We found that 21 functions used the input variables within. Since these functions deal with inputs

directly from external sources, we consider these functions to be critical and requires to be N-Versioned through our NVASA framework in order to improve the security of the overall web application. These 21 functions had an average of 14.2 LOC.

We then analyzed the Database queries within Moodle source code and as noted previously we found that the database queries were all located in 10 PHP pages from a total of 5061 PHP files (a %0.2). Therefore, applying the NVASA framework to these components wouldn't add significant overhead to the project. In addition, the decision algorithm will be replicated among the N-Versioned SQL components since it only compares the N outputs and reaches a consensus.

Since SQL injection attacks and Cross Site Scripting (XSS) are based on the language's escape/filter scheme, we suggest N-Versioning the query component function to filter and sanitize the input before committing or storing the input as shown in Figure 5.9. In addition, for ultimate security, we can apply the NVASA framework to the database to add another layer of security. This would be done by having N-Version databases. For example, MSSQL, MySQL, Oracle and/or PostgreSQL. To successfully N-Version the databases we will have to create a PHP component that acts as the vFork and vJoin. The vFork would translate the input to N-different queries based on the database model, MSSQL query, MySQL query or Oracle query, etc. Finally, the vJoin would run a decision algorithm on the databases' output to reach a consensus. Any exploited database, identified by the different output from the rest of the pool, would be forced to revert to the state prior to applying the query. Therefore, maintaining and protecting the data within the databases from any malicious queries or unauthorized access.

5.5. Guidelines and Policies

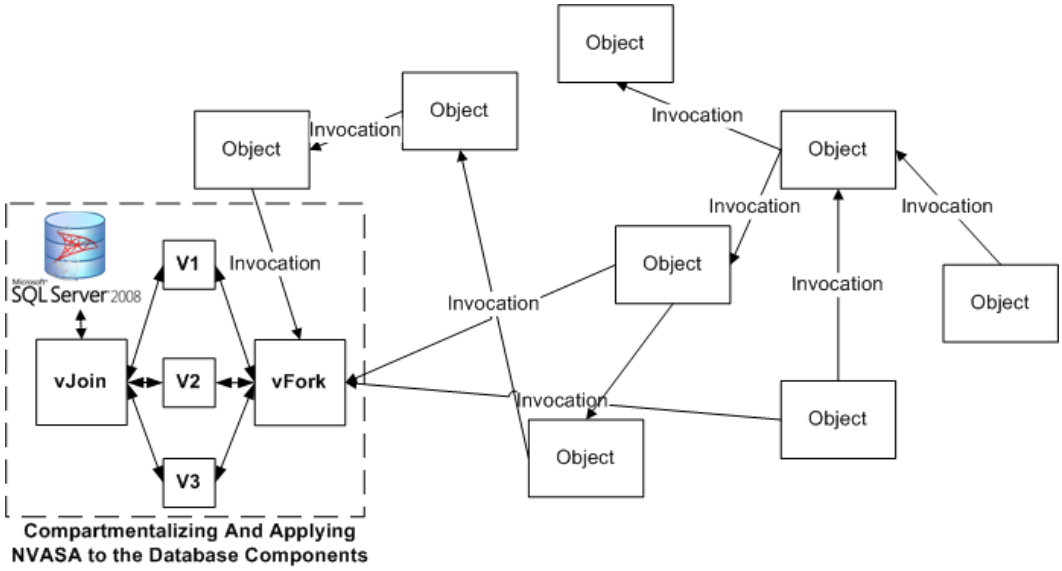

Figure 5.9. Applying the NVASA Framework to Moodle's Database Component

As we progressed with the experimental implementation, we researched best practices to configure the systems and network in the firmest and secure possible way. We prearranged the guidelines and policies in bullet points and divided them based on the systems components and architecture. First, we start with best practices in configuring the OS of the VMs. We highlighted the most important activities to be done to protect the instances within the cloud. Second, we highlight best practices in setting up the network and structure. Finally, we pin point best practices in code writing and software development.

5.5.1. Operating System

- The host machine and the virtual machines should run on secure operating systems. One suggestion is OpenBSD [106]. OpenBSD was built from the ground up to be secure and locked by default.

- The host machine and the virtual machine's OS should be updated frequently. This could be done easily with automated tools.
- The host machine and the VMs should be sandboxed and all unecessary ports should be closed [43].
- We highly recommend avoiding the usage of passwords to connect to the instances, instead using SSH keys [43].
- Periodic penetration testing should be conducted for the routing (interface) VM OS.

5.5.2. Network Configuration

- VMs running the NVASA framework and all layers should be placed in a separate virtual DMZ subnetwork than the organization's internal network.
- Any connection to internal servers within the internal network should be whitelisted, example database and backend servers.
- The connection between layers and VMs should be staticlly defined and whitelisted based on the IP, and MAC address.
- Messages between the NVASA layers and components must be encrypted. Transport Layer Security (SSL) protocol is one option.

5.5.3. Software Development

The NVASA framework reduces the security efforts during the development phases, but the interface component still needs to go through a strict security process and testing to avoid injecting any vulnerabilities during development.

- During the security evaluation the most important step is identifying the entry points of inputs to the program from users, applications, files, and external databases.
- Conduct unit-testing during the development of the routing layer application.
- Minimize using unsafe strings and buffer functions [61].
- Validate any external input by escaping all untrusted data and performing a whitelist validation.
- Use least privilege.
- Conduct periodic penetration testing for the routing layer application.
- Use of strong publicly known cryptograph algorithms.
- When writing the source code, avoid using insecure functions and libraries. One example is using the banned.h header file developed by Microsoft to support the SDL requirement to flag warnings when a dangerous function or library is used by the developer [7, 8].
- We highly recommend the usage of strong dynamic authentication schemes and avoid hard-coded credentials, particularly within client side applications [64].

5.6. Chapter Conclusion

In this Chapter, we conducted three experiments to validate our proposed NVASA architecture. Through the first two experiments, we learned that in fact the NVASA framework enhanced the application security through the decision algorithm. We also discovered from the AES implementation that even when all versions were successfully exploited, each version failed uniquely producing a different output. Therefore,

enabling the decision algorithm to detect and prevent the attack from taking place. Our third experimental revealed how compartmentalization is achieved through parsing tools to identify the critical components within the source code. We discovered that the input commands and database query code was a very small sub-set of the total lines of code (LOC), a total of %0.11. We also determined that on average, each input or database query LOC had 14 LOC within its function. Giving us roughly %1.6 of the total application to be N-Versioned to apply the NVASA framework to the critical components.

Through our experimental implementations, we prearranged a set of guidelines and policies that should be followed to implement a secure and successful NVASA framework application. In addition to these guidelines and policies, it is very crucial to define the implementation's guidelines, policies, and best practices related directly to its environment characteristics and transaction/data sensitivity. Misconfiguration is one of the top ten security issues within IT organizations and companies today [112]. Thus, it is very vital to follow these guidelines and policies and keep the configuration and setup of the system as secure as possible. It is also crucial to develop the NVASA framework components as protected and secure as possible. Keeping in mind the well-known vulnerabilities [112, 64]. Consequently, protect the framework from excessive DoS attacks.

Chapter 6

CONCLUSIONS AND FUTURE WORK

While application security is the source of most of the vulnerabilities that plague systems today, mitigation strategies have shown limited effects. The use of the N-Version Programming Methodology, traditionally a fault-tolerance approach, in distributed applications shows promise in strengthening application security in order to withstand both known and zero-day attacks. Therefore, in this praxis we propose the usage of the N-Version programming methodology to enhance application security through the implementation of our introduced N-Version Architecture Framework for Application Security (NVASA).

We started our praxis with a study of application security and why applications suffer from security vulnerabilities (bugs) primarily. In addition, we presented the various types of vulnerabilities and the phases they get injected into the application. These vulnerabilities are divided into three main categories: 1) architectural design, 2) implementation, and 3) operational and platform vulnerabilities. Moreover, we gave a brief study of the current related work in the field through different testing schemes and revision routines that are widely used nowadays. These schemes are effective in detecting some application vulnerabilities to some degree, but they fall short due to their complexity and the absence of any protection scheme against zero-day attacks.

Furthermore, we discussed the building blocks of our proposed four-layer NVASA framework. 1) The Routing Layer is the interface of the framework, it essentially routes the input to the N-Versions within the 2) N-Version Environment Layer to be executed. Each version executes concurrently sending the output to the 3) N-Version

Decision Layer where a consensus algorithm runs to reach a majority output or come to an agreement based on a unanimous output. This is defined beforehand based on the security level required.

We have presented how the NVASA architecture achieves better security in terms of detecting and preventing known attacks, while having a scheme to detect and prevent zero-day attacks as well. This is achieved by utilizing a decision algorithm within the N-Version decision layer that comes to a consensus based on all the N-Version outputs before applying any of these commands to the backend application server layer. Therefore, if a specific vulnerability was effective in exploiting one version the decision algorithm would still detect that attack and prevent it from propogating through the system. This proposed framework prevents application layer exploitation of the system which would lead to the destruction, modification, or leakage of the confidential information to malicious users.

However, applying the NVASA framework to large applications can introduce overhead and effort. Therefore, we discussed few overhead reduction schemes to reduce the overhead associated with implementing the NVASA framework. First reduction scheme is through compartmentalizing the application architecture into two or more components. Consequently, applying the NVASA framework exclusively to the critical components instead of the entire application. We have discussed two methods to identify the critical components. First, through static analysis schemes through source code parsing. Second, through expert security evaluators. Second reduction scheme is through the usage of various compilers of the same language to compile the source code and produce N-Versions of the application. Finally, through the usage of source-to-source language translators to implement N-Versions without having to program and administer each version through different groups of developers.

We have shown through our experiment work that web application architecture applying the NVASA Framework increases the resiliency and security posture of the exposed services. From experiment one, we tested the NVASA web system using a Buffer-Over-Flow input that was able to exploit only one version (the C++ version) from a total of three versions. However, the rest (the C and Java versions) revealed an error message to the decision layer. Therefore, enabling the decision algorithm from detecting the attack and preventing it from propagating through the system. In the same experiment, we tested the web system using a different attack using a Format-String input that was able to exploit a different version (the C version). Again, the remaining two versions (C++ and Java), revealed an error message. Thus, enabling the decision algorithm from detecting and preventing the attack from taking place.

In our second AES Cloud Implementation, we used three pre-developed implementations of AES to achieve the required diversity specified by the N-Version programming methodology. The N-Versions were written in C/C++, Java and C# .NET [110, 84, 88]. Each version was sandboxed in a separate VM. The routing and decision layers were developed using the Java language. The connection was achieved via web services. Furthermore, we tested the standalone versions against the NVASA AES implementation using a benchmark of 100 different inputs. With few inputs, we were able to exploit all versions, but fortunately, the NVASA framework detected and prevented the attack. This was ultimately achieved because each of the versions was exploited uniquely. Each version produced a unique output. Therefore, enabling the decision algorithm to detect the malformed input and prevented the exploit from propagating through the system.

Through our final experiment, we have statically analyzed a large open source web application called Moodle to identify the security critical parts. We parsed through the PHP source code looking for I/O and SQL queries. Moodle was constructed of

5061 PHP files and 1,060,585 lines of code. We discovered that the critical code (I/O and database query functions) accumulated a %1.6 of the total source code. Therefore, applying the NVASA framework to these components would add slight overhead to the development phases while increasing the security posture of the web application.

6.1. Future Work

6.1.1. Application Attacks

We have investigated some of the top application attacks in Chapter 3 and how the NVASA framework will detect and prevent these attacks from taking place and propagating through the system. Further study should be done to investigate other types of application attacks against the NVASA framework, and how the framework would still detect any discrepancy and prevent these attacks from taking place.

6.1.2. New Reduction Schemes

We have discussed few schemes to reduce the overhead associated with the implementation of the NVASA framework, future work should investigating new and improved schemes by improving on the proposed compartmentalization to identify the critical components, and paths within the applications. Hence, possibly applying the NVASA framework to the entire critical path for enhanced security. In addition, investigate if these proposed schemes are effective in detecting and preventing known attacks while still having a scheme to capture zero-day attacks. This could be done through a comparison between a full implemented NVASA framework application against the same application but compartmnetalized.

6.1.3. Automation of the NVASA Project

Further, we believe that most of the work performed to identify, and compartmentalize the critical components can be automated. Therefore, future work should focus on developing methodologies to automate the NVASA process to generate the architecture of the framework and versions. In addition, investigating the use of high-level modeling languages such as UML for the architectural definitions in conjunction with automated code generation.

6.1.4. Extra Protection Through a Buffer/Translator and a Learning Algorithm

Moreover, further work should focus on improving the routing layer and the communications between the routing layer and the N-Versions through a Buffer and a Translator. Furthermore, adding a learning algorithm to comprehend and update the translator with successful malicious inputs that succeeded in exploiting one or more versions. This will reduce any malicious inputs from penetrating the routing layer, if attempted again, and into the application. Consequently, improving the application's security, reducing any un-necessary overhead, and preventing any denial-of-service DoS attacks from affecting the performance of the application. Figure 6.1 shows a vision of the improved architecture. Besides, the data produced and collected by the learning algorithm could be used for further analysis to discover and identify new zero-day attacks.

6.1.5. NVASA framework to Protect the Client-Side from Zero-Day Attacks

In addition, we have discussed Cross-Site-Scripting attacks and how it affects both sides of the connection, the server-side and client-side. In this praxis, we focused on preventing attacks from taking place and breaching the server-side application. Future work should involve investigating the client-side by diversifying the browsers using

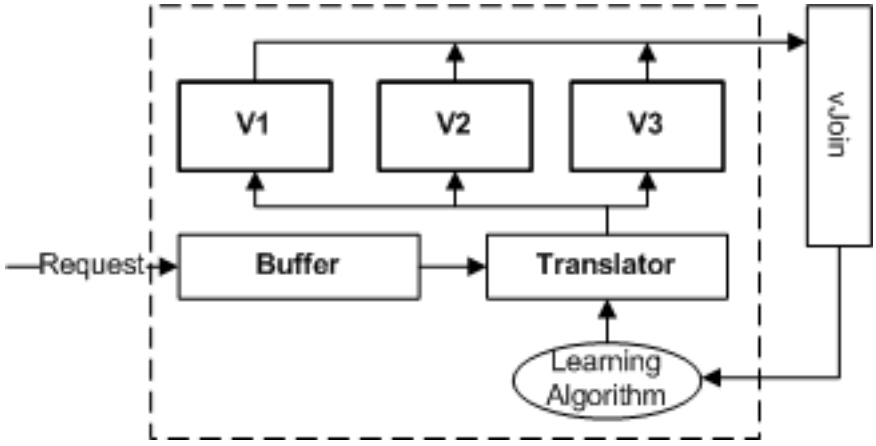

Figure 6.1. Vision of the Improved Routing Layer Communications within the NVASA Architecture

sandboxed versions and emulators to create an N-Versioned browser. Thus, detecting any malicious activity from harming users, whether the attack was XSS or a new zero-day exploit targeting the browser.

6.1.6. Automate Security for Cloud-Customers through a Specific NVASA Framework Model

Additionally, more work should investigate the implementation of a specific NVASA framework specialized to run on public clouds to automate security for cloud customers. The focus would be on creating a framework that receives the input and all defined set of restrictions, and values. Then run the input within a diverse set of emulators. These emulators would mimic diverse environments with various OS types and languages running on separate virtual machines. Therefore, capturing any un-usual behavior. This model would have to be specific to each attack, making it a central unit to detect and prevent known attacks. We should point out that such a specific system would not detect and prevent zero-day attacks, but will remove

the burden of detecting known-security attacks from the cloud customers. The same concept could be implemented by N-Versioning the anti-virus scanners to guarantee the detection of any malicious files uploaded to the cloud.

6.1.7. Economic Analysis of the NVASA Framework

N-Version programming is usually criticized for having high resource requirements, overhead, and implementation time. However, implementing the NVASA framework in distributed applications will improve the application security. Thus, avoiding huge economic expenses in security schemes and losses from successful breaches. For example, Sony predicted over $170 million in losses following the latest Play-Station Network hacking [37]. In addition, Citigroup has reported a loss of $2.7 million of customer's funds due to hacking activities [12]. These successful attacks on companies not only end up with financial losses, it damages the company's reputation and customer trust. Therefore, could be devastating in the long term. Future work should study the economical impact of implementing the NVASA framework and compare it to potential loss from historical security breaches.

Appendix A

SOURCE CODE

The source code of our AES NVASA implementation is divided into sections based on the architecture of the design. Section 1, includes the C/C++ implementation source code and the VM configuration as well as the web server (Apache) and CGI configuration. Section 2, includes the C# implementation and the VM configuration; in addition, the ASP.NET and IIS7 configurations. Finally, section 3 includes the Java AES implementation as well as the interface testing, routing and decision layer source code. Section 3 also includes the SOAP and CGI requests.

C/C++ AES Implementation

Configuring Apache and CGI

The C/C++ AES implementation ran on a Linux VM (Ubuntu). Therefore, we installed Apache as a web server and enabled CGI to run the C/C++ AES implementation as a web service.

1. Install Apache sudo apt-get install apache2
2. Setup Apache and CGI by creating a folder for the CGI and Perl scripts called cgi-bin under /var/www/
3. Set the permission to 750 to the cgi-bin folder
4. Added the below code to /etc/apache2/site-enabled/[DefaultFile]

```
<Directory ''/var/www/cgi-bin''>
        AllowOverride None
        Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
        AddHandler cgi-script cgi pl
        Order allow,deny
        Allow from all
 </Directory>
```

5. Included the executable CGI in cgi-bin.

ccode.cgi

```
#include <iostream>
#include <stdlib.h>
#include<string.h>
#include <stdio.h>

using namespace std;
#define Nb 4

//PlainText and KeyInput to get and store input
  string PlainText;
  string KeyInput;

// The number of rounds in AES Cipher.
//It is simply initiated to zero. The actual
//value is recieved in the program.
```

```
int Nr=12; //To encrypt using 192 bit Keys Only

// The number of 32 bit words in the key.
//It is simply initiated to zero. The actual
//value is recieved in the program.
int Nk=6; //To encrypt using 192 bit Keys Only

// in - it is the array that holds the plain text to be encrypted.
// out - it is the array that holds the key for encryption.
// state - the array that holds the intermediate results during encryption.
unsigned char in[16], out[16], state[4][4];

// The array that stores the round keys.
unsigned char RoundKey[240];

// The Key input to the AES Program
unsigned char Key[32];
string str;

int getSBoxValue(int num)
{
        int sbox[256] =   {
//0    1   2  3   4   5   6   7    8  9   A    B   C    D    E    F
0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76, //0
0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0, //1
0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15, //2
0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75, //3
0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84, //4
0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf, //5
0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8, //6
0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2, //7
0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73, //8
0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb, //9
0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79, //A
0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08, //B
0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a, //C
0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e, //D
0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf, //E
0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16 }; //F
return sbox[num];
}

// The round constant word array, Rcon[i], contains the values given by
// x to th e power (i-1) being powers of x (x is denoted as {02}) in the field GF(28)
// Note that i starts at 1, not 0).
int Rcon[255] = {
0x8d,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36,0x6c,0xd8,0xab,0x4d,0x9a,
0x2f,0x5e,0xbc,0x63,0xc6,0x97,0x35,0x6a,0xd4,0xb3,0x7d,0xfa,0xef,0xc5,0x91,0x39,
0x72,0xe4,0xd3,0xbd,0x61,0xc2,0x9f,0x25,0x4a,0x94,0x33,0x66,0xcc,0x83,0x1d,0x3a,
0x74,0xe8,0xcb,0x8d,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36,0x6c,0xd8,
0xab,0x4d,0x9a,0x2f,0x5e,0xbc,0x63,0xc6,0x97,0x35,0x6a,0xd4,0xb3,0x7d,0xfa,0xef,
0xc5,0x91,0x39,0x72,0xe4,0xd3,0xbd,0x61,0xc2,0x9f,0x25,0x4a,0x94,0x33,0x66,0xcc,
0x83,0x1d,0x3a,0x74,0xe8,0xcb,0x8d,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,
```

```
0x36,0x6c,0xd8,0xab,0x4d,0x9a,0x2f,0x5e,0xbc,0x63,0xc6,0x97,0x35,0x6a,0xd4,0xb3,
0x7d,0xfa,0xef,0xc5,0x91,0x39,0x72,0xe4,0xd3,0xbd,0x61,0xc2,0x9f,0x25,0x4a,0x94,
0x33,0x66,0xcc,0x83,0x1d,0x3a,0x74,0xe8,0xcb,0x8d,0x01,0x02,0x04,0x08,0x10,0x20,
0x40,0x80,0x1b,0x36,0x6c,0xd8,0xab,0x4d,0x9a,0x2f,0x5e,0xbc,0x63,0xc6,0x97,0x35,
0x6a,0xd4,0xb3,0x7d,0xfa,0xef,0xc5,0x91,0x39,0x72,0xe4,0xd3,0xbd,0x61,0xc2,0x9f,
0x25,0x4a,0x94,0x33,0x66,0xcc,0x83,0x1d,0x3a,0x74,0xe8,0xcb,0x8d,0x01,0x02,0x04,
0x08,0x10,0x20,0x40,0x80,0x1b,0x36,0x6c,0xd8,0xab,0x4d,0x9a,0x2f,0x5e,0xbc,0x63,
0xc6,0x97,0x35,0x6a,0xd4,0xb3,0x7d,0xfa,0xef,0xc5,0x91,0x39,0x72,0xe4,0xd3,0xbd,
0x61,0xc2,0x9f,0x25,0x4a,0x94,0x33,0x66,0xcc,0x83,0x1d,0x3a,0x74,0xe8,0xcb   };

// This function produces Nb(Nr+1) round keys.
The round keys are used in each round to encrypt the states.
void KeyExpansion()
{
        int i,j;
        unsigned char temp[4],k;

        // The first round key is the key itself.
        for(i=0;i<Nk;i++)
        {
                RoundKey[i*4]=Key[i*4];
                RoundKey[i*4+1]=Key[i*4+1];
                RoundKey[i*4+2]=Key[i*4+2];
                RoundKey[i*4+3]=Key[i*4+3];
        }

        // All other round keys are found from the previous round keys.
        while (i < (Nb * (Nr+1)))
        {
                                        for(j=0;j<4;j++)
                                        {
                                                temp[j]=RoundKey[(i-1) * 4 + j];
                                        }
                                        if (i \% Nk == 0)
                                        {
                                                // This function rotates the
                                                //4 bytes in a word to the left once.
                                                // [a0,a1,a2,a3] becomes [a1,a2,a3,a0]

                                                // Function RotWord()
                                                {
                                                        k = temp[0];
                                                        temp[0] = temp[1];
                                                        temp[1] = temp[2];
                                                        temp[2] = temp[3];
                                                        temp[3] = k;
                                                }

                                                // SubWord() is a function that takes
                                                //a four-byte input word and applies the
                                                //S-box to each of the four bytes to produce
                                                //an output word.
```

```
                                        // Function Subword()
                                        {
                                                temp[0]=getSBoxValue(temp[0]);
                                                temp[1]=getSBoxValue(temp[1]);
                                                temp[2]=getSBoxValue(temp[2]);
                                                temp[3]=getSBoxValue(temp[3]);
                                        }

                                        temp[0] =  temp[0] ^ Rcon[i/Nk];
                                }
                                else if (Nk > 6 && i \% Nk == 4)
                                {
                                        // Function Subword()
                                        {
                                                temp[0]=getSBoxValue(temp[0]);
                                                temp[1]=getSBoxValue(temp[1]);
                                                temp[2]=getSBoxValue(temp[2]);
                                                temp[3]=getSBoxValue(temp[3]);
                                        }
                                }
                                RoundKey[i*4+0] = RoundKey[(i-Nk)*4+0] ^ temp[0];
                                RoundKey[i*4+1] = RoundKey[(i-Nk)*4+1] ^ temp[1];
                                RoundKey[i*4+2] = RoundKey[(i-Nk)*4+2] ^ temp[2];
                                RoundKey[i*4+3] = RoundKey[(i-Nk)*4+3] ^ temp[3];
                                i++;
        }
}

// This function adds the round key to state.
// The round key is added to the state by an XOR function.
void AddRoundKey(int round)
{
        int i,j;
        for(i=0;i<4;i++)
        {
                for(j=0;j<4;j++)
                {
                        state[j][i] ^= RoundKey[round * Nb * 4 + i * Nb + j];
                }
        }
}

// The SubBytes Function Substitutes the values in the
// state matrix with values in an S-box.
void SubBytes()
{
        int i,j;
        for(i=0;i<4;i++)
        {
                for(j=0;j<4;j++)
                {
                        state[i][j] = getSBoxValue(state[i][j]);
```

```
                }
        }
}

// The ShiftRows() function shifts the rows in the state to the left.
// Each row is shifted with different offset.
// Offset = Row number. So the first row is not shifted.
void ShiftRows()
{
        unsigned char temp;

        // Rotate first row 1 columns to left
        temp=state[1][0];
        state[1][0]=state[1][1];
        state[1][1]=state[1][2];
        state[1][2]=state[1][3];
        state[1][3]=temp;

        // Rotate second row 2 columns to left
        temp=state[2][0];
        state[2][0]=state[2][2];
        state[2][2]=temp;

        temp=state[2][1];
        state[2][1]=state[2][3];
        state[2][3]=temp;

        // Rotate third row 3 columns to left
        temp=state[3][0];
        state[3][0]=state[3][3];
        state[3][3]=state[3][2];
        state[3][2]=state[3][1];
        state[3][1]=temp;
}

// xtime is a macro that finds the product of {02} and the argument to xtime modulo {1b}
#define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))

// MixColumns function mixes the columns of the state matrix
void MixColumns()
{
        int i;
        unsigned char Tmp,Tm,t;
        for(i=0;i<4;i++)
        {
                t=state[0][i];
                Tmp = state[0][i] ^ state[1][i] ^ state[2][i] ^ state[3][i] ;
                Tm = state[0][i] ^ state[1][i] ; Tm = xtime(Tm); state[0][i] ^= Tm ^ Tmp ;
                Tm = state[1][i] ^ state[2][i] ; Tm = xtime(Tm); state[1][i] ^= Tm ^ Tmp ;
                Tm = state[2][i] ^ state[3][i] ; Tm = xtime(Tm); state[2][i] ^= Tm ^ Tmp ;
                Tm = state[3][i] ^ t ; Tm = xtime(Tm); state[3][i] ^= Tm ^ Tmp ;
        }
}
```

```
// Cipher is the main function that encrypts the PlainText.
void Cipher()
{
        int i,j,round=0;

        //Copy the input PlainText to state array.
        for(i=0;i<4;i++)
        {
                for(j=0;j<4;j++)
                {
                        state[j][i] = in[i*4 + j];
                }
        }

        // Add the First round key to the state before starting the rounds.
        AddRoundKey(0);

        // There will be Nr rounds.
        // The first Nr-1 rounds are identical.
        // These Nr-1 rounds are executed in the loop below.
        for(round=1;round<Nr;round++)
        {
                SubBytes();
                ShiftRows();
                MixColumns();
                AddRoundKey(round);
        }

        // The last round is given below.
        // The MixColumns function is not here in the last round.
        SubBytes();
        ShiftRows();
        AddRoundKey(Nr);

        // The encryption process is over.
        // Copy the state array to output array.
        for(i=0;i<4;i++)
        {
                for(j=0;j<4;j++)
                {
                        out[i*4+j]=state[j][i];
                }
        }
}

int main()
{
  char* data = getenv("QUERY_STRING");
  string s = data;
  int i;

 // receiving and parsing the browser input to extract the Key and PlainText
```

```
// This process was taken from various sites and is
// known to be vulnerable due to lack of filtration

 cout << "Content-type: text/plain \n\n";
 //Delete variable names and signs
 s.erase(0,4);
 int positionPT = s.find("&PlainText=", 0);
 s.replace(positionPT,11," ");//Replace 11 Charecters from PositionPT with a Space
 int positionSp = s.find(" ", 0); // Mark the space between the Key and PlainText

 KeyInput = s.substr(0,positionSp); //Extract the Key
 PlainText = s.substr((positionSp+1), s.length()); //Extract the PlainText

 //Debugging purposes Show Key nd PlainText
 //cout << "Key: \"" << KeyInput<<"\""<<endl;
 //cout << "PlainText: \"" << PlainText<<"\""<<endl;

 int m;
 int Mark=0;
 int number=0;
 int KeyLength=KeyInput.length()-1;
 //Debugging purposes
 //printf("\n\n\n\n");
 char temp[KeyInput.length()];
 strcpy(temp, KeyInput.c_str());
 ////Debugging purposes
 //printf("\n\n\n\n This is Temp: \"\%s\" \n",temp);
 for(m=0;m<KeyLength;m=m+2)
 {

        sscanf(&temp[m],"\%02x", &number);
        Key[Mark]=(unsigned char)number;
        //printf("\%02x, and Key[\%d]=\%02x\n",number, Mark,Key[Mark]);
        Mark++;
 }
 Mark=0;
 char temp2[PlainText.length()];
 strcpy(temp2, PlainText.c_str());
 int PlainTextLength=PlainText.length()-1;
 for(m=0;m<PlainTextLength;m=m+2)
 {
        sscanf(&temp2[m],"\%02x", &number);
        in[Mark]=(unsigned char)number;
        Mark++;
 }

 // The KeyExpansion routine must be called before encryption.
 KeyExpansion();

 // The next function call encrypts the PlainText
        // with the Key using AES algorithm.
 Cipher();
```

```
    // Output the encrypted text.
    //printf("\nCipher Text:\n");
    for(i=0;i<Nb*4;i++)
    {
            printf("\%02x",out[i]);
    }
    printf("\n\n");

    return 0;
}
```

C# AES Implementation

Configuring IIS7 and ASP.NET

The C# AES implementation ran on a Windows7 VM. Therefore, we installed IIS7 as a web server and enabled ASP.NET to run the C# AES implementation as a web service.

1. Install IIS7 and ASP.NET by going to Control Panel, then click on Programs. Choose "Turn Windows features on or off". Enable IIS7 and ASP.NET features.
2. Publish the C# web service to a new folder C:/AES.
3. Set the permission of the C:/AES to allow Everyone to Read and Execute.
4. In the IIS management tool, create a new Web Site and point the physical file to C:/AES and set a port. In this implementation we used 65534 and 5791.
5. Enable Directory Browsing within the IIS management tool.
6. In Windows firewall, open the specified ports.

AES.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace AesLib
{
  public class Aes  // Advanced Encryption Standard
  {
    public enum KeySize { Bits128, Bits192, Bits256 };
                // key size, in bits, for construtor

    private int Nb;          // block size in 32-bit words.  Always 4 for AES.  (128 bits).
    private int Nk;          // key size in 32-bit words.  4, 6, 8.  (128, 192, 256 bits).
    private int Nr;          // number of rounds. 10, 12, 14.
```

```
private byte[] key;      // the seed key. size will be 4 * keySize from ctor.
private byte[,] Sbox;    // Substitution box
private byte[,] iSbox;   // inverse Substitution box
private byte[,] w;       // key schedule array.
private byte[,] Rcon;    // Round constants.
private byte[,] State;   // State matrix

public Aes(KeySize keySize, byte[] keyBytes)
{
  SetNbNkNr(keySize);

  this.key = new byte[this.Nk * 4];  // 16, 24, 32 bytes
  keyBytes.CopyTo(this.key, 0);

  BuildSbox();
  BuildInvSbox();
  BuildRcon();
  KeyExpansion();  // expand the seed key into a key schedule and store in w

}  // Aes constructor

public void Cipher(byte[] input, byte[] output)  // encipher 16-bit input
{
  // state = input
  this.State = new byte[4,Nb];  // always [4,4]
  for (int i = 0; i < (4 * Nb); ++i)
  {
    this.State[i \% 4, i / 4] = input[i];
  }

  AddRoundKey(0);

  for (int round = 1; round <= (Nr - 1); ++round)  // main round loop
  {
    SubBytes();
    ShiftRows();
    MixColumns();
    AddRoundKey(round);
  }  // main round loop

  SubBytes();
  ShiftRows();
  AddRoundKey(Nr);

  // output = state
  for (int i = 0; i < (4 * Nb); ++i)
  {
    output[i] = this.State[i \% 4, i / 4];
  }

}  // Cipher()
```

```
public void InvCipher(byte[] input, byte[] output)  // decipher 16-bit input
{
  // state = input
  this.State = new byte[4,Nb];  // always [4,4]
  for (int i = 0; i < (4 * Nb); ++i)
  {
    this.State[i \% 4, i / 4] = input[i];
  }

  AddRoundKey(Nr);

  for (int round = Nr-1; round >= 1; --round)  // main round loop
  {
    InvShiftRows();
    InvSubBytes();
    AddRoundKey(round);
    InvMixColumns();
  }  // end main round loop for InvCipher

  InvShiftRows();
  InvSubBytes();
  AddRoundKey(0);

  // output = state
  for (int i = 0; i < (4 * Nb); ++i)
  {
    output[i] = this.State[i \% 4, i / 4];
  }

}  // InvCipher()

private void SetNbNkNr(KeySize keySize)
{
  this.Nb = 4;      // block size always = 4 words = 16 bytes = 128 bits for AES

  if (keySize == KeySize.Bits128)
  {
    this.Nk = 4;    // key size = 4 words = 16 bytes = 128 bits
    this.Nr = 10;   // rounds for algorithm = 10
  }
  else if (keySize == KeySize.Bits192)
  {
    this.Nk = 6;    // 6 words = 24 bytes = 192 bits
    this.Nr = 12;
  }
  else if (keySize == KeySize.Bits256)
  {
    this.Nk = 8;    // 8 words = 32 bytes = 256 bits
    this.Nr = 14;
  }
}  // SetNbNkNr()

private void BuildSbox()
```

```
    {
      this.Sbox = new byte[16,16] {  // populate the Sbox matrix
/* 0     1    2    3    4    5    6    7    8    9    a    b    c    d    e    f */
/*0*/  {0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76},
/*1*/  {0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0},
/*2*/  {0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15},
/*3*/  {0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75},
/*4*/  {0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84},
/*5*/  {0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf},
/*6*/  {0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8},
/*7*/  {0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2},
/*8*/  {0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73},
/*9*/  {0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb},
/*a*/  {0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79},
/*b*/  {0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08},
/*c*/  {0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a},
/*d*/  {0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e},
/*e*/  {0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf},
/*f*/  {0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16} };

    } // BuildSbox()

    private void BuildInvSbox()
    {
      this.iSbox = new byte[16,16] {  // populate the iSbox matrix
/* 0     1    2    3    4    5    6    7    8    9    a    b    c    d    e    f */
/*0*/  {0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb},
/*1*/  {0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb},
/*2*/  {0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e},
/*3*/  {0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25},
/*4*/  {0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92},
/*5*/  {0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84},
/*6*/  {0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06},
/*7*/  {0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b},
/*8*/  {0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73},
/*9*/  {0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e},
/*a*/  {0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b},
/*b*/  {0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4},
/*c*/  {0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f},
/*d*/  {0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef},
/*e*/  {0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61},
/*f*/  {0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d} };

    } // BuildInvSbox()

    private void BuildRcon()
    {
      this.Rcon = new byte[11,4] { {0x00, 0x00, 0x00, 0x00},
                                   {0x01, 0x00, 0x00, 0x00},
                                   {0x02, 0x00, 0x00, 0x00},
                                   {0x04, 0x00, 0x00, 0x00},
                                   {0x08, 0x00, 0x00, 0x00},
                                   {0x10, 0x00, 0x00, 0x00},
```

```
                                {0x20, 0x00, 0x00, 0x00},
                                {0x40, 0x00, 0x00, 0x00},
                                {0x80, 0x00, 0x00, 0x00},
                                {0x1b, 0x00, 0x00, 0x00},
                                {0x36, 0x00, 0x00, 0x00} };
} // BuildRcon()

private void AddRoundKey(int round)
{

  for (int r = 0; r < 4; ++r)
  {
    for (int c = 0; c < 4; ++c)
    {
      this.State[r,c] = (byte) ( (int)this.State[r,c] ^ (int)w[(round*4)+c,r] );
    }
  }
} // AddRoundKey()

private void SubBytes()
{
  for (int r = 0; r < 4; ++r)
  {
    for (int c = 0; c < 4; ++c)
    {
      this.State[r,c] = this.Sbox[ (this.State[r,c] >> 4), (this.State[r,c] & 0x0f) ];
    }
  }
} // SubBytes

private void InvSubBytes()
{
  for (int r = 0; r < 4; ++r)
  {
    for (int c = 0; c < 4; ++c)
    {
      this.State[r,c] = this.iSbox[ (this.State[r,c] >> 4), (this.State[r,c] & 0x0f) ];
    }
  }
} // InvSubBytes

private void ShiftRows()
{
  byte[,] temp = new byte[4,4];
  for (int r = 0; r < 4; ++r)  // copy State into temp[]
  {
    for (int c = 0; c < 4; ++c)
    {
      temp[r,c] = this.State[r,c];
    }
  }

  for (int r = 1; r < 4; ++r)  // shift temp into State
```

```
  {
    for (int c = 0; c < 4; ++c)
    {
      this.State[r,c] = temp[ r, (c + r) \% Nb ];
    }
  }
}  // ShiftRows()

private void InvShiftRows()
{
  byte[,] temp = new byte[4,4];
  for (int r = 0; r < 4; ++r)  // copy State into temp[]
  {
    for (int c = 0; c < 4; ++c)
    {
      temp[r,c] = this.State[r,c];
    }
  }
  for (int r = 1; r < 4; ++r)  // shift temp into State
  {
    for (int c = 0; c < 4; ++c)
    {
      this.State[r, (c + r) \% Nb ] = temp[r,c];
    }
  }
}  // InvShiftRows()

private void MixColumns()
{
  byte[,] temp = new byte[4,4];
  for (int r = 0; r < 4; ++r)  // copy State into temp[]
  {
    for (int c = 0; c < 4; ++c)
    {
      temp[r,c] = this.State[r,c];
    }
  }

  for (int c = 0; c < 4; ++c)
  {
    this.State[0,c] = (byte) ( (int)gfmultby02(temp[0,c]) ^ (int)gfmultby03(temp[1,c]) ^
                               (int)gfmultby01(temp[2,c]) ^ (int)gfmultby01(temp[3,c]) );
    this.State[1,c] = (byte) ( (int)gfmultby01(temp[0,c]) ^ (int)gfmultby02(temp[1,c]) ^
                               (int)gfmultby03(temp[2,c]) ^ (int)gfmultby01(temp[3,c]) );
    this.State[2,c] = (byte) ( (int)gfmultby01(temp[0,c]) ^ (int)gfmultby01(temp[1,c]) ^
                               (int)gfmultby02(temp[2,c]) ^ (int)gfmultby03(temp[3,c]) );
    this.State[3,c] = (byte) ( (int)gfmultby03(temp[0,c]) ^ (int)gfmultby01(temp[1,c]) ^
                               (int)gfmultby01(temp[2,c]) ^ (int)gfmultby02(temp[3,c]) );
  }
}  // MixColumns

private void InvMixColumns()
{
```

```
  byte[,] temp = new byte[4,4];
  for (int r = 0; r < 4; ++r)  // copy State into temp[]
  {
    for (int c = 0; c < 4; ++c)
    {
      temp[r,c] = this.State[r,c];
    }
  }

  for (int c = 0; c < 4; ++c)
  {
    this.State[0,c] = (byte) ( (int)gfmultby0e(temp[0,c]) ^ (int)gfmultby0b(temp[1,c]) ^
                               (int)gfmultby0d(temp[2,c]) ^ (int)gfmultby09(temp[3,c]) );
    this.State[1,c] = (byte) ( (int)gfmultby09(temp[0,c]) ^ (int)gfmultby0e(temp[1,c]) ^
                               (int)gfmultby0b(temp[2,c]) ^ (int)gfmultby0d(temp[3,c]) );
    this.State[2,c] = (byte) ( (int)gfmultby0d(temp[0,c]) ^ (int)gfmultby09(temp[1,c]) ^
                               (int)gfmultby0e(temp[2,c]) ^ (int)gfmultby0b(temp[3,c]) );
    this.State[3,c] = (byte) ( (int)gfmultby0b(temp[0,c]) ^ (int)gfmultby0d(temp[1,c]) ^
                               (int)gfmultby09(temp[2,c]) ^ (int)gfmultby0e(temp[3,c]) );
  }
}  // InvMixColumns

private static byte gfmultby01(byte b)
{
  return b;
}

private static byte gfmultby02(byte b)
{
  if (b < 0x80)
    return (byte)(int)(b <<1);
  else
    return (byte)( (int)(b << 1) ^ (int)(0x1b) );
}

private static byte gfmultby03(byte b)
{
  return (byte) ( (int)gfmultby02(b) ^ (int)b );
}

private static byte gfmultby09(byte b)
{
  return (byte)( (int)gfmultby02(gfmultby02(gfmultby02(b))) ^
                 (int)b );
}

private static byte gfmultby0b(byte b)
{
  return (byte)( (int)gfmultby02(gfmultby02(gfmultby02(b))) ^
                 (int)gfmultby02(b) ^
                 (int)b );
}
```

```
private static byte gfmultby0d(byte b)
{
  return (byte)( (int)gfmultby02(gfmultby02(gfmultby02(b))) ^
                 (int)gfmultby02(gfmultby02(b)) ^
                 (int)(b) );
}

private static byte gfmultby0e(byte b)
{
  return (byte)( (int)gfmultby02(gfmultby02(gfmultby02(b))) ^
                 (int)gfmultby02(gfmultby02(b)) ^
                 (int)gfmultby02(b) );
}

private void KeyExpansion()
{
  this.w = new byte[Nb * (Nr+1), 4];  // 4 columns of bytes corresponds to a word

  for (int row = 0; row < Nk; ++row)
  {
    this.w[row,0] = this.key[4*row];
    this.w[row,1] = this.key[4*row+1];
    this.w[row,2] = this.key[4*row+2];
    this.w[row,3] = this.key[4*row+3];
  }

  byte[] temp = new byte[4];

  for (int row = Nk; row < Nb * (Nr+1); ++row)
  {
    temp[0] = this.w[row-1,0]; temp[1] = this.w[row-1,1];
    temp[2] = this.w[row-1,2]; temp[3] = this.w[row-1,3];

    if (row \% Nk == 0)
    {
      temp = SubWord(RotWord(temp));

      temp[0] = (byte)( (int)temp[0] ^ (int)this.Rcon[row/Nk,0] );
      temp[1] = (byte)( (int)temp[1] ^ (int)this.Rcon[row/Nk,1] );
      temp[2] = (byte)( (int)temp[2] ^ (int)this.Rcon[row/Nk,2] );
      temp[3] = (byte)( (int)temp[3] ^ (int)this.Rcon[row/Nk,3] );
    }
    else if ( Nk > 6 && (row \% Nk == 4) )
    {
      temp = SubWord(temp);
    }

    // w[row] = w[row-Nk] xor temp
    this.w[row,0] = (byte) ( (int)this.w[row-Nk,0] ^ (int)temp[0] );
    this.w[row,1] = (byte) ( (int)this.w[row-Nk,1] ^ (int)temp[1] );
    this.w[row,2] = (byte) ( (int)this.w[row-Nk,2] ^ (int)temp[2] );
    this.w[row,3] = (byte) ( (int)this.w[row-Nk,3] ^ (int)temp[3] );
```

```
  }  // for loop
}  // KeyExpansion()

private byte[] SubWord(byte[] word)
{
  byte[] result = new byte[4];
  result[0] = this.Sbox[ word[0] >> 4, word[0] & 0x0f ];
  result[1] = this.Sbox[ word[1] >> 4, word[1] & 0x0f ];
  result[2] = this.Sbox[ word[2] >> 4, word[2] & 0x0f ];
  result[3] = this.Sbox[ word[3] >> 4, word[3] & 0x0f ];
  return result;
}

private byte[] RotWord(byte[] word)
{
  byte[] result = new byte[4];
  result[0] = word[1];
  result[1] = word[2];
  result[2] = word[3];
  result[3] = word[0];
  return result;
}

public  void Dump()
{
  Console.WriteLine("Nb = " + Nb + " Nk = " + Nk + " Nr = " + Nr);
  Console.WriteLine("\nThe key is \n" + DumpKey() );
  Console.WriteLine("\nThe Sbox is \n" + DumpTwoByTwo(Sbox));
  Console.WriteLine("\nThe w array is \n" + DumpTwoByTwo(w));
  Console.WriteLine("\nThe State array is \n" + DumpTwoByTwo(State));
}

public string DumpKey()
{
  string s = "";
  for (int i = 0; i < key.Length; ++i)
    s += key[i].ToString("x2") + " ";
  return s;
}

public string DumpTwoByTwo(byte[,] a)
{
  string s ="";
  for (int r = 0; r < a.GetLength(0); ++r)
  {
    s += "["+r+"]" + " ";
    for (int c = 0; c < a.GetLength(1); ++c)
    {
      s += a[r,c].ToString("x2") + " " ;
    }
    s += "\n";
  }
  return s;
```

```
    }

  }  // class Aes
}  // ns AesLib
```

Service.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Services;
using System.IO;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
// To allow this Web Service to be called
//from script, using ASP.NET AJAX, uncomment the following line.
// [System.Web.Script.Services.ScriptService]

public class Service : System.Web.Services.WebService
{
    System.Text.ASCIIEncoding encoding = new System.Text.ASCIIEncoding();
    public Service () {

        //Uncomment the following line if using designed components
        //InitializeComponent();
    }

    [WebMethod]
    public string AES_Interface(String Key, String PlainText) {

        byte[] plainText = StringToByteArray(PlainText);
        byte[] cipherText = new byte[16];
        byte[] decipheredText = new byte[16];

        byte[] keyBytes = StringToByteArray(Key);
        AesLib.Aes a = new AesLib.Aes(AesLib.Aes.KeySize.Bits192, keyBytes);
        //Create an Object for the C# AES Implementation

        a.Cipher(plainText, cipherText);
        //Call Cipher to Encrypt the Plain Text
        DisplayAsBytes(cipherText);
        //Save the result into a file called CSOutput.txt

        //a.InvCipher(cipherText, decipheredText);
        //Uncomment this to Decrypt the Cipher Text

        string text = System.IO.File.ReadAllText(@"C:\AES\CSOutput.txt");
        //Read the Cipher Text from the CSOutput.txt File
```

```
        return text;
        //Return the Cipher Text to the Requester
    }

    static void DisplayAsBytes(byte[] bytes)
    {
        System.IO.File.Delete(@"C:\AES\CSOutput.txt");

        for (int i = 0; i < bytes.Length; ++i)
        {
            System.IO.File.AppendAllText(@"C:\AES\CSOutput.txt", (bytes[i].ToString("x2")));
            //Console.Write(bytes[i].ToString("x2") + " ");
            if (i > 0 && i \% 16 == 0)
                System.IO.File.AppendAllText(@"C:\AES\CSOutput.txt", "\n");
        }
        System.IO.File.AppendAllText(@"C:\AES\CSOutput.txt", "");
    }  // DisplayAsBytes()

    public static byte[] StringToByteArray(string hex)
    {
        return Enumerable.Range(0, hex.Length)
                         .Where(x => x \% 2 == 0)
                         .Select(x => Convert.ToByte(hex.Substring(x, 2), 16))
                         .ToArray();
    }

}
```

Java Routing and Decision Layers

```
package testnvasaversions;
import java.io.*;
import java.net.*;
import java.io.FileInputStream;
import java.util.Iterator;
import java.util.Vector;

/**
 *
 * @author MajidMalaika
 */
public class TestNVASAVersions {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        String Key = "";
        String PlainText = "";
        String C_Version = "";
        String Java_Version = "";
        String CS_Version = "";
        int consensus = 0;
        TestNVASAVersions tv = new TestNVASAVersions();
        StringBuilder contents = new StringBuilder();

        try {
            //use buffering, reading one line at a time
            //FileReader always assumes default encoding is OK!
            String aFile = "C:\\AES\\pwd_pt2.txt";
            BufferedReader input = new BufferedReader(new FileReader(aFile));
            try {
                String line = null; //not declared within while loop
                /*
                 * readLine is a bit quirky :
                 * it returns the content of a line MINUS the newline.
                 * it returns null only for the END of the stream.
                 * it returns an empty String if two newlines appear in a row.
                 */
                while ((line = input.readLine()) != null) {
                    String[] st = line.split(":");
                    Key = st[0];
                    PlainText = st[1];
                    /* Call the C# Version with the given Key and PlainText
                     * and store the result in CS_Version String */
                    CS_Version = tv.CS_Version(Key, PlainText);
                    //System.out.println("C# Version Output: \"" +
                   //          CS_Version + "\"");
```

```
                /* Call the Java Version with the given Key and PlainText
                 * and store the
                result in Java_Version String */
                Java_Version = tv.Java_Version(Key, PlainText);
               // System.out.println("Java Version Output: \"" +
                //        Java_Version + "\"");

                /* Call the C/C++ Version with the given Key and PlainText
                 * and store the result in C_Version String */
                C_Version = tv.CPP_Version(Key, PlainText);
                //System.out.println("C/C++ Version Output: \"" +
                //        C_Version + "\"");

                // NVASA Decision Maker
                consensus = tv.NVASA_Decision(CS_Version,
                        Java_Version, C_Version);
                //if(consensus < 2)
                System.out.println("Key: " + Key +
                        ", PlainText: " + PlainText + ", Consensus: "
                        + consensus + " and the version's output is "
                        + "as follow, C: "+C_Version + ", J: "+
                        Java_Version + ", C#: " + CS_Version);
            }
        } finally {
            input.close();
        }
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

private String CS_Version(String Key, String PT) {

    SoapRequestBuilder s = new SoapRequestBuilder();
    s.Server = "129.119.107.191";
    //Server IP address, for localhost I use 127.0.0.1
    s.MethodName = "AES_Interface";
    s.XmlNamespace = "http://tempuri.org/";
    s.WebServicePath = "/Service.asmx";
    s.SoapAction = s.XmlNamespace + s.MethodName;
    s.Key = Key;
    s.PlainText = PT;
    String response = s.sendRequest();

    return response;
}

private String Java_Version(String Key, String PT) {
    String strln = new String();
    try {
        // Create two files one for the plain text and one for the key
        // these two files would work as the feeders to the Versions
```

```
            FileWriter fstreamPT = new FileWriter(
                    "C:\\AES\\plaintextWrite.txt");
            FileWriter fstreamK = new FileWriter("C:\\AES\\keyWrite.txt");
            BufferedWriter outPT = new BufferedWriter(fstreamPT);
            BufferedWriter outK = new BufferedWriter(fstreamK);
            outPT.write(PT);
            outK.write(Key);
            //Close the output stream
            outPT.close();
            outK.close();
        } catch (Exception e) {//Catch exception if any
            System.err.println("Error: " + e.getMessage());
        }

        //Encrypting the User PlainText using the User's Key
        //Testing Implementation 1 Java

        //Read the PT and K from the 2 created files
        GetBytes getInput = new GetBytes("C:\\AES\\plaintextWrite.txt", 16);
        byte[] in = getInput.getBytes();
        GetBytes getKey = new GetBytes("C:\\AES\\keyWrite.txt", 24);
        byte[] key = getKey.getBytes();
        AESencrypt aes = new AESencrypt(key, 6);
        //Print.printArray("Plaintext:      ", in);
        //Print.printArray("Key:            ", key);
        byte[] out = new byte[16];
        aes.Cipher(in, out);
        //Print.printArray("Ciphertext:     ", out);

        //Print the Cipher Text to C:\\AES\\OutPutAES.txt
        // This is defined in the Print Class
        Print.printArrayToFile("", out);

        try {
            FileInputStream fstream = new FileInputStream(
                    "C:\\AES\\OutPutAES.txt");
            DataInputStream inCipher = new DataInputStream(fstream);
            BufferedReader br = new BufferedReader(
                    new InputStreamReader(inCipher));
            strln = br.readLine();
        } catch (Exception e) {//Catch exception if any
            System.err.println("Error: " + e.getMessage());
        }
        strln = strln.replace(" ", "");
        return strln;

    }

    private String CPP_Version(String Key, String PT) {

        String inputLine = "";
        try {
```

```
            URL c_URL = new URL("http://129.119.107.182/cgi-bin/ccode.cgi?Key="
                    + Key + "&PlainText=" + PT);
            URLConnection yc = c_URL.openConnection();
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(
                    yc.getInputStream()));
            inputLine = in.readLine();
            in.close();
        } catch (Exception e) {
            System.out.println("Error with C/C++ Version: "
                    + e);
        }
        return inputLine;
    }

    private int NVASA_Decision(String CS, String Java, String CPP){

        int num = 0;
        if(Java.equals(CS))
            ++num;
        if(Java.equals(CPP))
            ++num;

        return num;
    }
}
```

Java AES Implementation

The Java AES implementation and the routing and decision layer ran on a Windows 7 VM. Therefore, we didn't need to expose the Java implementation as a web service, a procedure call was sufficient in this case, but since the routing and decision layer needed to communicate with the other N-Versions, we needed to create a SOAP request to be able to connect to the C# .NET version (Shown in SoapRequestBuilder.java) and needed to create a URL request to connect to the C/C++ version (Shown in the TESTNVASAVersions.java).

AESEncrypt.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package testnvasaversions;

public class AESencrypt {
   private final int Nb = 4; // words in a block, always 4 for now
   private int Nk; // key length in words
   private int Nr; // number of rounds, = Nk + 6
   private int wCount; // position in w for RoundKey (= 0 each encrypt)
   private AEStables tab; // all the tables needed for AES
   private byte[] w; // the expanded key

   // AESencrypt: constructor for class.  Mainly expands key
   public AESencrypt(byte[] key, int NkIn) {
      Nk = NkIn; // words in a key, = 4, or 6, or 8
      Nr = Nk + 6; // corresponding number of rounds
      tab = new AEStables(); // class to give values of various functions
      w = new byte[4*Nb*(Nr+1)]; // room for expanded key

      KeyExpansion(key, w);

   }

   // Cipher: actual AES encrytion
   public void Cipher(byte[] in, byte[] out) {
      wCount = 0; // count bytes in expanded key throughout encryption
      byte[][] state = new byte[4][Nb]; // the state array
```

```
    Copy.copy(state, in); // actual component-wise copy
    AddRoundKey(state); // xor with expanded key
    for (int round = 1; round < Nr; round++) {
        //Print.printArray("Start round  " + round + ":", state);
        SubBytes(state); // S-box substitution
        ShiftRows(state); // mix up rows
        MixColumns(state); // complicated mix of columns
        AddRoundKey(state); // xor with expanded key
    }
    //Print.printArray("Start round " + Nr + ":", state);
    SubBytes(state); // S-box substitution
    ShiftRows(state); // mix up rows
    AddRoundKey(state); // xor with expanded key
    Copy.copy(out, state);
}

// KeyExpansion: expand key, byte-oriented code, but tracks words
private void KeyExpansion(byte[] key, byte[] w) {
    byte[] temp = new byte[4];
    // first just copy key to w
    int j = 0;
    while (j < 4*Nk) {
       w[j] = key[j++];
    }
    // here j == 4*Nk;
    int i;
    while(j < 4*Nb*(Nr+1)) {
        i = j/4; // j is always multiple of 4 here
        // handle everything word-at-a time, 4 bytes at a time
        for (int iTemp = 0; iTemp < 4; iTemp++)
            temp[iTemp] = w[j-4+iTemp];
        if (i \% Nk == 0) {
            byte ttemp, tRcon;
            byte oldtemp0 = temp[0];
            for (int iTemp = 0; iTemp < 4; iTemp++) {
                if (iTemp == 3) ttemp = oldtemp0;
                else ttemp = temp[iTemp+1];
                if (iTemp == 0) tRcon = tab.Rcon(i/Nk);
                else tRcon = 0;
                temp[iTemp] = (byte)(tab.SBox(ttemp) ^ tRcon);
            }
        }
        else if (Nk > 6 && (i\%Nk) == 4) {
            for (int iTemp = 0; iTemp < 4; iTemp++)
                temp[iTemp] = tab.SBox(temp[iTemp]);
        }
        for (int iTemp = 0; iTemp < 4; iTemp++)
            w[j+iTemp] = (byte)(w[j - 4*Nk + iTemp] ^ temp[iTemp]);
        j = j + 4;
    }
}

// SubBytes: apply Sbox substitution to each byte of state
```

```
    private void SubBytes(byte[][] state) {
        for (int row = 0; row < 4; row++)
            for (int col = 0; col < Nb; col++)
                state[row][col] = tab.SBox(state[row][col]);
    }

    // ShiftRows: simple circular shift of rows 1, 2, 3 by 1, 2, 3
    private void ShiftRows(byte[][] state) {
        byte[] t = new byte[4];
         for (int r = 1; r < 4; r++) {
            for (int c = 0; c < Nb; c++)
                t[c] = state[r][(c + r)\%Nb];
            for (int c = 0; c < Nb; c++)
                state[r][c] = t[c];
        }
    }

    // MixColumns: complex and sophisticated mixing of columns
    private void MixColumns(byte[][] s) {
        int[] sp = new int[4];
        byte b02 = (byte)0x02, b03 = (byte)0x03;
        for (int c = 0; c < 4; c++) {
            sp[0] = tab.FFMul(b02, s[0][c]) ^ tab.FFMul(b03, s[1][c]) ^
                                    s[2][c]  ^                 s[3][c];
            sp[1] =                 s[0][c]  ^ tab.FFMul(b02, s[1][c]) ^
                    tab.FFMul(b03, s[2][c]) ^                  s[3][c];
            sp[2] =                 s[0][c]  ^                 s[1][c]  ^
                    tab.FFMul(b02, s[2][c]) ^ tab.FFMul(b03, s[3][c]);
            sp[3] = tab.FFMul(b03, s[0][c]) ^                  s[1][c]  ^
                                    s[2][c]  ^ tab.FFMul(b02, s[3][c]);
            for (int i = 0; i < 4; i++) s[i][c] = (byte)(sp[i]);
        }
    }

    // AddRoundKey: xor a portion of expanded key with state
    private void AddRoundKey(byte[][] state) {
        for (int c = 0; c < Nb; c++)
            for (int r = 0; r < 4; r++)
                state[r][c] = (byte)(state[r][c] ^ w[wCount++]);
    }
}
```

AEStables.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
```

```
package testnvasaversions;
// AEStables: construct various 256-byte tables needed for AES
public class AEStables {
   public AEStables() {
      loadE(); loadL(); loadInv();
      loadS(); loadInvS(); loadPowX();
   }

   private byte[] E = new byte[256]; // "exp" table (base 0x03)
   private byte[] L = new byte[256]; // "Log" table (base 0x03)
   private byte[] S = new byte[256]; // SubBytes table
   private byte[] invS = new byte[256]; // inverse of SubBytes table
   private byte[] inv = new byte[256]; // multiplicative inverse table
   private byte[] powX = new byte[15]; // powers of x = 0x02

   // Routines to access table entries
   public byte SBox(byte b) {
      return S[b & 0xff];
   }

   public byte invSBox(byte b) {
      return invS[b & 0xff];
   }

   public byte Rcon(int i) {
      return powX[i-1];
   }

   // FFMulFast: fast multiply using table lookup
   public byte FFMulFast(byte a, byte b){
      int t = 0;;
      if (a == 0 || b == 0) return 0;
      t = (L[(a & 0xff)] & 0xff) + (L[(b & 0xff)] & 0xff);
      if (t > 255) t = t - 255;
      return E[(t & 0xff)];
   }

   // FFMul: slow multiply, using shifting
   public byte FFMul(byte a, byte b) {
      byte aa = a, bb = b, r = 0, t;
      while (aa != 0) {
         if ((aa & 1) != 0)
            r = (byte)(r ^ bb);
         t = (byte)(bb & 0x80);
         bb = (byte)(bb << 1);
         if (t != 0)
            bb = (byte)(bb ^ 0x1b);
         aa = (byte)((aa & 0xff) >> 1);
      }
      return r;
   }

   // loadE: create and load the E table
```

```
private void loadE() {
   byte x = (byte)0x01;
   int index = 0;
   E[index++] = (byte)0x01;
   for (int i = 0; i < 255; i++) {
      byte y = FFMul(x, (byte)0x03);
      E[index++] = y;
      x = y;
   }
}

// loadL: load the L table using the E table
private void loadL() { // careful: had 254 below several places
   int index;
   for (int i = 0; i < 255; i++) {
       L[E[i] & 0xff] = (byte)i;
   }
}

// loadS: load in the table S
private void loadS() {
   int index;
   for (int i = 0; i < 256; i++)
       S[i] = (byte)(subBytes((byte)(i & 0xff)) & 0xff);
}

// loadInv: load in the table inv
private void loadInv() {
   int index;
   for (int i = 0; i < 256; i++)
       inv[i] = (byte)(FFInv((byte)(i & 0xff)) & 0xff);
}

// loadInvS: load the invS table using the S table
private void loadInvS() {
   int index;
   for (int i = 0; i < 256; i++) {
       invS[S[i] & 0xff] = (byte)i;
   }
}

// loadPowX: load the powX table using multiplication
private void loadPowX() {
   int index;
   byte x = (byte)0x02;
   byte xp = x;
   powX[0] = 1; powX[1] = x;
   for (int i = 2; i < 15; i++) {
       xp = FFMul(xp, x);
       powX[i] = xp;
   }
}
```

```
    // FFInv: the multiplicative inverse of a byte value
    public byte FFInv(byte b) {
        byte e = L[b & 0xff];
        return E[0xff - (e & 0xff)];
    }

    // ithBIt: return the ith bit of a byte
    public int ithBit(byte b, int i) {
        int m[] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};
        return  (b & m[i]) >> i;
    }

    // subBytes: the subBytes function
    public int subBytes(byte b) {
        byte inB = b;
        int res = 0;
        if (b != 0) // if b == 0, leave it alone
            b = (byte)(FFInv(b) & 0xff);
        byte c = (byte)0x63;
        for (int i = 0; i < 8; i++) {
            int temp = 0;
            temp = ithBit(b, i) ^ ithBit(b, (i+4)\%8) ^ ithBit(b, (i+5)\%8) ^
              ithBit(b, (i+6)\%8) ^ ithBit(b, (i+7)\%8) ^ ithBit(c, i);
            res = res | (temp << i);
        }
        return res;
    }
}
```

Copy.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package testnvasaversions;
public class Copy {
    private static final int Nb = 4;
    // copy: copy in to state
    public static void copy(byte[][] state, byte[] in) {
        int inLoc = 0;
        for (int c = 0; c < Nb; c++)
            for (int r = 0; r < 4; r++)
                state[r][c] = in[inLoc++];
    }

    // copy: copy state to out
    public static void copy(byte[] out, byte[][] state) {
```

```
        int outLoc = 0;
        for (int c = 0; c < Nb; c++)
            for (int r = 0; r < 4; r++)
                out[outLoc++] = state[r][c];
    }
}
//The class Print prints 1-and 2-dimensional arrays of bytes for debugging:
```

GetBytes.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package testnvasaversions;

// GetBytes: fetch array of bytes, represented in hex
import java.io.*;
public class GetBytes {
    private String fileName; // input filename
    private int arraySize; // number of bytes to read
    private Reader in;

    // GetBytes: constructor, opens input file
    public GetBytes(String file, int n) {
        fileName = file;
        arraySize = n;
        try {
            in = new FileReader(fileName);
        } catch (IOException e) {
            System.out.println("Exception opening " + fileName);
        }
    }

    // getNextChar: fetches next char
    private char getNextChar() {
        char ch = ' '; // = ' ' to keep compiler happy
        try {
            ch = (char)in.read();
        } catch (IOException e) {
            System.out.println("Exception reading character");
        }
        return ch;
    }

    // val: return int value of hex digit
    private int val(char ch) {
```

```
        if (ch >= '0' && ch <= '9') return ch - '0';
        if (ch >= 'a' && ch <= 'f') return ch - 'a' + 10;
        if (ch >= 'A' && ch <= 'F') return ch - 'A' + 10;
        return -1000000;
    }

    // getBytes: fetch array of bytes in hex
    public byte[] getBytes() {
        byte[] ret = new byte[arraySize];
        for (int i = 0; i < arraySize; i++) {
            char ch1 = getNextChar();
            char ch2 = getNextChar();
            ret[i] = (byte)(val(ch1)*16 + val(ch2));
        }
        return ret;
    }
}
```

Print.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package testnvasaversions;
import java.io.*;

// Print: print arrays of bytes
public class Print {
    private static final int Nb = 4;
    private static String[] dig = {"0","1","2","3","4","5","6","7",
                                   "8","9","a","b","c","d","e","f"};

    // hex: print a byte as two hex digits
    public static String hex(byte a) {
        return dig[(a & 0xff) >> 4] + dig[a & 0x0f];
    }

    public static void printArray(String name, byte[] a) {
        System.out.print(name + " ");
        for (int i = 0; i < a.length; i++)
            System.out.print(hex(a[i]) + " ");
        System.out.println();
    }

    public static void printArrayToFile(String name, byte[] a) {
        try{
```

```
        // Create file

          FileWriter fstreamPT = new FileWriter("C:\\AES\\OutPutAES.txt");
          BufferedWriter outPT = new BufferedWriter(fstreamPT);

        //Close the output stream

         outPT.write(name + " ");
         for (int i = 0; i < a.length; i++)
         outPT.write(hex(a[i]) + " ");
         outPT.write('\n');
         outPT.close();
        }catch (Exception e){//Catch exception if any
             System.err.println("Error: " + e.getMessage());
         }
   }

   public static void printArray(String name, byte[][] s) {
      System.out.print(name + " ");
      for (int c = 0; c < Nb; c++)
         for (int r = 0; r < 4; r++)
            System.out.print(hex(s[r][c]) + " ");
      System.out.println();
   }
}
```

SoapRequestBuilder.java

```
/*
 * Majid Malaika
 * This Request Builder was taken from David S Hobbs and was modified
 * to fit the NVASA Implementation for AES Encryption.
 * Link Provided in References.
 */
package testnvasaversions;
import java.applet.Applet;
import java.awt.*;
import java.net.*;
import java.util.*;
import java.io.*;
/**
 *
 * @author MajidMalaika
 */
class SoapRequestBuilder {
  String Server = "";
  String WebServicePath = "";
  String SoapAction = "";
```

```
String MethodName = "";
String XmlNamespace = "";
String Key = "";
String PlainText = "";
private Vector ParamNames = new Vector();
private Vector ParamData = new Vector();

public void AddParameter(String Name, String Data) {
  ParamNames.addElement( (Object) Name);
  ParamData.addElement( (Object) Data);
}

public String sendRequest() {
  String retval = "";
  Socket socket = null;
  try {
    socket = new Socket(Server, 5791);//53671);
  }
  catch (Exception ex1) {
    return ("Error: "+ex1.getMessage());
  }

  try {
    OutputStream os = socket.getOutputStream();
    boolean autoflush = true;
    PrintWriter out = new PrintWriter(socket.getOutputStream(), autoflush);
    BufferedReader in = new BufferedReader(new InputStreamReader(socket.
        getInputStream()));
    //Here I calculate the Length to be associated with the SOAP Request
    // The length depends on the number of Char of the entire request
    // In my Case its 314 (295) for all the static parts plus the Key and Plain
    //Text lengths
    int length = 295 + Server.length() + (MethodName.length() * 2) +
            XmlNamespace.length();
     length+= Key.length();
     length+= PlainText.length();
    // send an HTTP request to the web service

    out.println("POST " + WebServicePath + " HTTP/1.1");
    out.println("Host: " + Server);
    out.println("Content-Type: text/xml; charset=utf-8");
    out.println("Content-Length: " + String.valueOf(length));
    out.println("SOAPAction: \"" + "http://tempuri.org/AES_Interface" + "\"");
    out.println();

    out.println("<?xml version=\"1.0\" encoding=\"utf-8\"?>");
    out.println("<soap:Envelope xmlns:xsi=\"http://www.w3.org/2001/XML"
            + "Schema-instance\" xmlns:xsd=\"http://www.w3.org/2001/"
            + "XMLSchema\" xmlns:soap=\"http://schemas.xmlsoap.org/soap/"
            + "envelope/\">");
    out.println("<soap:Body>");
    out.println("<" + MethodName + " xmlns=\"" + XmlNamespace + "\">");
    out.println("<Key>"+Key+"</Key>");
```

```
      out.println("<PlainText>"+PlainText+"</PlainText>");
      out.println("</" + MethodName + ">");
      out.println("</soap:Body>");
      out.println("</soap:Envelope>");
      out.println();


      // Read the response from the server ... times out if the response takes
      // more than 3 seconds
      String inputLine;
      StringBuffer sb = new StringBuffer(5000);
      int wait_seconds = 3;
      boolean timeout = false;
      long m = System.currentTimeMillis();
      inputLine = in.readLine();
      while (inputLine != null && !timeout) {
          sb.append(inputLine + "\n");
          out.println("a");
        if ( (System.currentTimeMillis() - m) > (1000 * wait_seconds))
            timeout = true;
        inputLine = in.readLine();
      }
      in.close();

      // The StringBuffer sb now contains the complete result from the
      // webservice in XML format.  You can parse this XML if you want to
      // get more complicated results than a single value.
      if (!timeout) {
          String returnparam = MethodName + "Result";
        int start = sb.toString().indexOf("<" + returnparam + ">") +
            returnparam.length() + 2;
        int end = sb.toString().indexOf("</" + returnparam + ">");
        //Extract a singe return parameter
        retval = sb.toString().substring(start, end);
      }
      else {
        retval="Error: response timed out.";
      }
      socket.close();
    }
    catch (Exception ex) {
      return ("Error: cannot communicate. " + ex);
    }

    return retval;
  }
}
```

Java Source Code Parser

NVASA_Parser.java

```
package nvasa_parser;

import java.io.*;
import java.util.Hashtable;

public class NVASA_Parser {
    static int LOC = 0;
    static int LOCC = 0;
    static int NOF = 0;
    static String criticlCode = "fopen";
    static String FileDirectory="C:\\MoodleFiles\\Moodle_files_"
            +criticlCode+".txt";

    public static void main(String[] args) {

        //directory is the folder holding the source code to be parsed
        File directory = new File("C:\\moodle");
        //Moodle_Files.txt contains all unique files that contain the
        //critical code.
        File f = new File(FileDirectory);
        File files[] = directory.listFiles();
        String[] list = directory.list();
        //criticalCode contains the critical command we're searching for.

      if(f.exists())
          f.delete();
      NVASA_Parser np = new NVASA_Parser();
      np.recur("C:\\moodle",criticlCode);
      System.out.println("************************************************");
      System.out.println("Total PHP Files Parsed: "+NOF);
      System.out.println("Total lines of code: "+LOC);
      System.out.println("Total lines of \"" + criticlCode + "\" = " +LOCC);
      System.out.println("Unique Files containing \"" + criticlCode + "\" ="+
              np.numOfFiles(FileDirectory));
      System.out.println("************************************************");
      System.out.println("Unique files within gathered so far: "
              +np.totalUniqFiles());
    }

    public void recur(String Directory, String ccode){

        if(Directory.contains(".php")){
            try{
              // Open the file that is the first
              // command line parameter
              NOF++;
              FileInputStream fstream =
                      new FileInputStream(Directory);
              // Get the object of DataInputStream
              DataInputStream in = new DataInputStream(fstream);
              BufferedReader br = new BufferedReader(new InputStreamReader(in));
              String strLine,tempString;
```

```
            //Read File Line By Line
            while ((strLine = br.readLine()) != null){
            // Print the content on the console
            //System.out.println (strLine);
               tempString = strLine.replaceAll("\\s+", "");
               if(!tempString.startsWith("//")){
                        //System.out.println(tempString);
                 LOC++;
                  if(strLine.contains(ccode)){
                     LOCC++;
                      if(!checkForStringInFile(Directory))
                      writeToFile(Directory);
                  }
               }
            }
      }catch(Exception e){}
    }
    else if(Directory.contains(".")){
        //Ignore other types of files
    }
    else{
        try{
            File directory = new File(Directory);
            File files[] = directory.listFiles();
            for(File f : files){
                this.recur(f.toString(),ccode);
            }
        }catch(NullPointerException e){}
    }
}
public void writeToFile(String st){

    try{
      FileWriter wstream = new
              FileWriter(FileDirectory,true);
      BufferedWriter out= new
              BufferedWriter(wstream);
      out.write(st+"\n");
      out.close();
    }catch(Exception e){System.out.println(e);}
}
public boolean checkForStringInFile(String strLine){

    boolean flag = false;
    try{
          FileInputStream fstream =
                  new FileInputStream(FileDirectory);
          DataInputStream in = new DataInputStream(fstream);
          BufferedReader br = new BufferedReader(new InputStreamReader(in));
          String sstrLine;
          //Read File Line By Line
          while ((sstrLine = br.readLine()) != null){
              if(sstrLine.contains(strLine))
```

```
                    return true;
                }
                }catch(Exception e){}
        return false;
    }
    private int numOfFiles(String Directory){
        int i=0;
      try{
          FileInputStream fstream =
                  new FileInputStream(Directory);
          // Get the object of DataInputStream
          DataInputStream in = new DataInputStream(fstream);
          BufferedReader br = new BufferedReader(new InputStreamReader(in));
          String strLine,tempString;
          //Read File Line By Line
          while ((strLine = br.readLine()) != null){
              i++;
          }
        }catch(Exception e){}
        return i;
    }

    private int totalUniqFiles(){

        Hashtable ht = new Hashtable();
        int i=0;

        try{
            File directory = new File("C:\\MoodleFiles\\");
            File files[] = directory.listFiles();
            for(File f : files){
            try{
                FileInputStream fstream =
                  new FileInputStream(f);
                  // Get the object of DataInputStream
                  DataInputStream in = new DataInputStream(fstream);
                  BufferedReader br = new BufferedReader(new
                          InputStreamReader(in));
                  String strLine,tempString;
                  //Read File Line By Line
                  while ((strLine = br.readLine()) != null){

                      if(!ht.containsKey(strLine.hashCode())){
                          i++;
                          ht.put(strLine.hashCode(), strLine);
                      }
                  }
                }catch(Exception e){}
                }
            }catch(NullPointerException e){}
        return i;
    }
}
```

REFERENCES

[1] Ada programming language,
http://www.wordiq.com/definition/Ada_programming_language.

[2] Android developing community,
http://developer.android.com/index.html.

[3] Antivirus bypass
http://www.offensive-security.com/metasploit-unleashed/Antivirus_Bypass.

[4] Apache on ubuntu: Installing a web server,
http://www.easy-ubuntu-linux.com/apache-ubuntu-install.html.

[5] Apple fixes 130 mac os x vulnerabilities.

[6] Appsamuck free iphone apps development community,
http://appsamuck.com/.

[7] Banned c/c++ functions,
http://www.microsoft.com/download/en/details.aspx?id=24817.

[8] Banned.h c/c++ functions tutorial,
http://msdn.microsoft.com/en-us/security/Video/gg675008.

[9] Buffer-over-flow,
https://www.owasp.org/index.php/Buffer_overflow.

[10] C to java translator,
http://download.cnet.com/C-To-Java-Converter/3000-2213_4-10080009.html.

[11] Can a canary solve the buffer overflow problem?
http://www.networkworld.com/newsletters/sec/0830sec1.html.

[12] Citigroup admits 2.7 million dollars of customers money stolen due to hack
http://techland.time.com/2011/06/27/citigroup-admits-2-7-million\-of-customers-money-lost-due-to-hack/.

[13] Cross site scripting (xss) cheat sheet,
http://ha.ckers.org/xss.html.

[14] The cross-site scripting (xss) faq,
http://www.cgisecurity.com/xss-faq.html.

[15] Cross-site scripting (xss),
https://www.owasp.org/index.php/Top_10_2010-A2.

[16] Eucalyptus open source cloud,
http://www.eucalyptus.com/.

[17] Foundations: What are buffer overflows?
http://www.watchguard.com/infocenter/editorial/135136.asp.

[18] The gnu prolog (prolog to c translator),
http://www.gprolog.org/.

[19] How and why to use parameterized queries,
http://blogs.msdn.com/b/sqlphp/archive/2008/09/30/
how-and-why-to-use-parameterized-queries.aspx.

[20] How to develop a simple iphone app submit it to itunes,
http://www.makeuseof.com/tag/develop-simple-iphone-app\
-submit-itunes/.

[21] How to install apache2 webserver with php,cgi and perl support in ubuntu server,
http://www.ubuntugeek.com/how-to-install-apache2-webserver-with\
-phpcgi-and-perl-support-in-ubuntu-server.html.

[22] How to: Protect from sql injection in asp.net,
http://msdn.microsoft.com/en-us/library/ff648339.aspx.

[23] Html input types used by php,
http://www.w3schools.com/html/html_forms.asp.

[24] Java to c translator,
http://www.cs.arizona.edu/projects/sumatra/toba/doc/.

[25] Java to paython translator,
http://code.google.com/p/java2python/.

[26] Moodle a course management system (cms)
http://moodle.org/.

[27] Nagios the industry standard in it infrastructure monitoring, http://www.nagios.org/.

[28] Openxenmanager graphical interface to manage xen cloud platform, http://sourceforge.net/projects/openxenmanager/.

[29] Penetration testing reading room, sans institute, http://www.sans.org/reading_room/whitepapers/testing/.

[30] Php file upload, http://www.w3schools.com/php/php_file_upload.asp.

[31] Php sql injection, http://en.wikibooks.org/wiki/PHP_Programming/SQL_Injection.

[32] Preventing sql injection in java, https://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java.

[33] Prolog cafe (prolog to java translator), http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/.

[34] Rollback mysql, http://dev.mysql.com/doc/refman/5.0/en/commit.html.

[35] Rollback transaction microsoft sql server, http://msdn.microsoft.com/en-us/library/ms181299.aspx.

[36] Securityfocus bypassing filters, http://www.securityfocus.com/archive/1/archive/1/514939/100/0/threaded.

[37] Sony estimates 171 million dollar loss due to psn hack http://gengame.net/2011/05/sony-estimates-171-million-dollar-\loss-due-to-psn-hack/.

[38] Sql injection - are parameterized queries safe?, http://taylorza.blogspot.com/2009/04/sql-injection-are-\parameterized-queries.html.

[39] Sql injection cheat sheet esp: for filter evasion, http://ha.ckers.org/sqlinjection/.

[40] Sql injection cheat sheet, http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/.

[41] Sql stored procedures,
http://msdn.microsoft.com/en-us/library/aa174792(v=sql.80).aspx.

[42] Stored procedure,
http://en.wikipedia.org/wiki/Stored_procedure.

[43] Tips for securing your ec2 instance,
http://aws.amazon.com/articles/1233.

[44] Total virus multiple anti-virus analysis tool,
http://www.virustotal.com/.

[45] Ubuntu enterprise cloud computing,
http://www.ubuntu.com/cloud.

[46] Ubuntu enterprise cloud computing server guide,
https://help.ubuntu.com/10.04/serverguide/C/uec.html.

[47] Ubuntu linux openssh server installation and configuration,
http://www.cyberciti.biz/faq/ubuntu-linux-openssh-server-\
installation-and-configuration/.

[48] Unrestricted file upload,
https://www.owasp.org/index.php/Unrestricted_File_Upload.

[49] Why file upload forms are a major security threat,
http://www.acunetix.com/websitesecurity/upload-forms-threat.htm.

[50] Xen private cloud platform,
http://xen.org/products/cloudxen.html.

[51] Cyber security: A crisis of prioritization19. Tech. rep., Presidents Information Technology Advisory Committee, 2005.

[52] 2008 internet crime report. Tech. rep., The National White Collar Crime Center Annual Report, 2008.

[53] Application security and development. Tech. rep., DISA for the DoD, 2008.

[54] Security threat report,
http://viewer.media.bitpipe.com/978207627_109/1286217697_648/
sophossecuritythreatreportmidyear2010wpna.pdf. Tech. rep., SOPHOS, 2010.

[55] 359R, I. J. S. W. . N. Programming languages - guide for the use of the ada. Tech. rep., National White Collar Crime Center, 2001.

[56] Alliance, C. S. Security guidance for critical areas of focus in cloud computing. Tech. rep., Cloud Security Alliance Security Guidance, V2.1, 2009.

[57] Anley, C. Advanced sql injection in sql server applications. An NGSSoftware Insight Security Research (NISR).

[58] Avizienis, A. *The Methodology of N-version Programming*. John Wiley and Sons, 1995.

[59] Ayewah, N., Hovemeyer, D., Morgenthaler, J., Penix, J., and Pugh, W. Using static analysis to find bugs.

[60] Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[61] Belk, M., Coles, M., Goldschmidt, C., Howard, M., Randolph, K., Saario, M., Sondhi, R., Tarandach, I., Vh-Sipil, A., and Yonchev, Y. Fundamental practices for secure software development. Tech. rep., SAFECode, 2011.

[62] Brain, R. Bypassing asp .net "validaterequest" for script injection attacks. Tech. rep., SAFECode, 2008.

[63] Brilliant, S., Knight, J., and Leveson, N. Analysis of faults in an n-version software experiment.

[64] Brown, M. Cwe/sans top 25 most dangerous software errors of 2010.

[65] Buecker, A., Ashley, A., Borrett, M., Lu, M., Muppidi, S., and Readshaw, N. Understanding soa security design and implementation. Tech. rep., IBM, 2007.

[66] Catteddu, D., and Massonet, P. Cloud computing, benefits, risks and recommendations. Information Security European Network and Information Security.

[67] CCMB. Common criteria information technology security evaluation. Tech. rep., CCMB, 2009.

[68] Cowan, C., Wagle, P., Pu, C., Beattie, S., and Walpole, J. Buffer overflows: Attacks and defenses for the vulnerability of the decade. System Administration and Network Security (SANS).

[69] Developers, C. Code review at cisco systems. Tech. rep., Cisco Systems Inc. Labs, June 2006.

[70] Eden, A., and Hirshfeld, Y. Principles in formal specification of object oriented design and architecture. Centre for Advanced Studies on Collaborative research.

[71] Eigler, F. Mudflap: Pointer use checking for c/c++. Tech. rep., In Proc. of the GCC Developers Summit, 2003.

[72] Etoh, Hiroaki, and Yoda, K. Protecting from stack-smashing attacks. Tech. rep., IBM Labs, 2000.

[73] Fong, E., and Okun, V. Web application scanners: Definitions and functions. Proceedings of the 40th Hawaii International Conference on System Sciences.

[74] Garlan, D., and Shaw, M. An introduction to software architecture.

[75] Grossman, J. Vulnerability assessment plus web application firewall (va+waf).

[76] IC3. 2009 internet crime report. Tech. rep., National White Collar Crime Center, 2009.

[77] Jourdan, G. Command injection. Tech. rep., 2005.

[78] Kaner, C., Falk, J., and Nguyen, H. *Testing Computer Software*, second ed. Wiley, 1999.

[79] Kissel, R., Stine, K., Scholl, M., Rossman, H., Fahlsing, J., and Gulick, J. Security considerations in the system development life cycle. NIST Special Publication 800-64 Revision 2.

[80] Knight, J. Detection of faults and software reliability analysis. Tech. rep., NASA Publication, 1986.

[81] Labs, S. M. C. Cloud computing, 2008.

[82] Lyu, M., and Avizienis, A. Assuring design diversity in n-version software: A design paradigm for n-version programming. PROC. DCCA.

[83] Malaika, M., Nair, S., and Coyle, F. Application security automation for cloud computing. CloudComp.

[84] McCaffrey, J. Keep your data secure with the new advanced encryption standard, 2003.

[85] Messmer, E. Security of virtualization, cloud computing divides it and security pros.

[86] Miller, J. Overview whitepaper of application security testing methodologies. BlackHat, 2009.

[87] Netinant, P., Elrad, T., and Fayad, M. A layered approach to building open aspect-oriented systems: a framework for the design of on-demand system demodularization.

[88] Niyaz, P. Advanced encryption standard c/c++.

[89] Northcutt, S., Shenk, J., Shackleford, D., Rosenberg, T., Siles, R., and Mancini, S. Penetration testing: Assessing your overall security before attackers do. Tech. rep., SANS Institute, 2006.

[90] Olzak, T. Web application security - buffer overflows: Are you really at risk?

[91] Rabah, K. Building your own private clouds using ubuntu and eucalyptus, `http://www.docstoc.com/docs/40802356/Build-your-Own-Private-\ Clouds-using-Ubuntu-1004-Eucalyptus-Enterprise-Cloud-Computing-\ Platform`. Tech. rep.

[92] Rae, A., and Fidge, C. Identifying critical components during information security evaluation. Australian Computer Society Inc.

[93] Sachitano, A., Chapman, R., and Hamilton, J. Security in software architecture: A case study, 2004.

[94] Schneier, B. *Secrets and Lies*. John Wiley and Sons, New York, NY, 2000.

[95] Schneier, B. The commercial speech arms race, October 2009.

[96] Schneier, B. Software monoculture, December 2010.

[97] scut / team TESO Hacking Group. Exploiting format string vulnerabilities. Tech. rep., 2001.

[98] Seltzer, L. Apple releases vast os x security update.

[99] Sha, L. Using simplicity to control complexity. 0740-7459/01 IEEE Software.

[100] Singh, A., Sinha, N., Agrawal, N. N. L., and Princeton). Avatars pennies: Cheap n-version programming for replication. HotDep.

[101] Sotirov, A. *Automatic Vulnerability Detection Using Static Source Code Analysis Thesis*. PhD thesis, 2005.

[102] Sutton, Greene, A., and Amini, P. Fuzzing: Brute force vulnerability discovery. Recon.

[103] Team, E. D. Amazon web services: Overview of security processes. Tech. rep., Amazon, 2009.

[104] Team, E. D. Security best practices. Tech. rep., Amazon, 2010.

[105] Team, I. D. Ibm point of view: Security and cloud computing. Tech. rep., IBM, 2009.

[106] Vaughan-Nichols, S. Openbsd: The most secure os around, http://www.zdnet.com/news/openbsd-the-most-secure-os-around/298564.

[107] Viega, J., and McGraw, G. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, New York, NY, 2001.

[108] Virtualization, S. S. Cloud computing virtualization benefits, 2006.

[109] W3Schools. Long term trends of operating system usage, a statistical analysis. Tech. rep., W3Schools, 2010.

[110] Wagner, N. Laws of cryptography: Java code for aes encryption, 2001.

[111] WhiteHat. Whitehat website security statistic report. Tech. rep., WhiteHat Quarterly Security Report, 9th Edition, 2010.

[112] Williams, J., Wicher, D., and Alamri, L. Owasp top 10 for 2010 released will you help us reach every web developer in the world?