4. ADAPTIVE VULNERABILITY ANALYSIS ENHANCED WITH MUTATION TESTING

4.1 Introduction

Over the years, several researches have attempted to apply methodologies originally developed for software testing and software quality assessment to perform Software Security Assessment (Aslam, 1995; Spafford, 1990; Beizer, 1983; Voas, Ghosh, McGraw, Charron & Miller, 1996). Especially, researchers in the domain of failure-tolerance and reliable software have found that the problem in Computer Security is a special case of failure tolerance, where software failure is the failure of a system to enforce the security policies defined for the system (Voas et al., 1996). This lead Voas et al. to adapt a technique called “Extended Propagation Analysis”, originally used in assessing safety-critical software (Voas & Miller, 1995; Voas & Miller, 1994) and develop a dynamic software analysis algorithm called “Adaptive Vulnerability Analysis”.

Voas et al. have worked on adapting a technique called “Extended Propagation Analysis”, originally meant for assessment of safety-critical software (Voas & Miller, 1995; Voas & Miller, 1994) to perform Software Security Assessment (Voas et al., 1996). They define a Security Attack as a “dynamic event that occurs during the execution of a piece of software”. According to them, a Vulnerability comprises 2 parts: a potential defect or weakness in an information system together with the knowledge required to exploit the defect. Voas et. al. caution that AVA is intended to provide a relative measure of Software Security. Their approach allows system vendors to know apriori whether their systems are secure against a known class of threats $T=\{t_1, t_2, \ldots, t_n\}$ and this set $T$ can be extended with the discovery of new intrusion schemes. Though AVA attempts to simulate both novel threats and
recurrent threats the success achieved with novel threats is less certain than those achieved with recurrent threats.

4.1.1 The State of Security Assessment

Voas et al. paint a picture of the current state of Security Assessment, which is essentially a “penetrate and patch” approach. A site is tested by applying security attacks and if the result of any attack leads to an intrusion, a patch is prepared and installed on the site. This patch prevents any further exploitation of the vulnerability that caused the attack. But this is any way does not guard against exploitation of other vulnerabilities in the Software. Computer Emergency Response Teams (CERT’s) are set up to coordinate the installation of patches for known vulnerabilities but following the “if it is not broken, don’t fix it” paradigm, many organizations choose to ignore CERT alerts.

Most organizations resort to the Tiger Team Penetration Testing approach which is a form of Stress Testing where a group of experts attempt to break an installed system. The disadvantages of such an approach are obvious. Different Testing Teams have differing levels of ability to detect vulnerabilities and the technique essentially reveals the presence of security flaws and not their absence. Deriving metrics for quantifying Security from the results of Tiger Team Penetration Testing is far from being statistically re-demonstrable (Voas et al., 1996).

Voas et al. claim that models such as TCSEC and SSE CMM suffer from the weakness of product versus process verification. They go on to claim that all these models have to do with the process the software undergoes during its development and it is possible for an organization to jump through all the right loops resulting in an insecure product. Automation of Tiger team testing is supported by a host of tools ranging from COPS (Farmer & Venema, 1993), Tripwire (Kim & Spafford, 1993),
ISS (Klaus, 1995). The disadvantage of using these tools is that the interpretation of the results of the testing is highly subjective.

In addition to these attempts to deal with security of the software post-installation, there are firewalls which suffer from the drawback of being unable to deal with flaws in higher-level protocol and applications and Data Encryption which suffers from the problem of flawed implementation of Encryption Algorithms. In fact, in the National Information Systems Security Conference held at 1995, panelists agreed that encryption does nothing in the handling of buggy software (Spafford, 1996).

Having highlighted the inadequacies of the existing state of security assessment, Voas et al. propose a method that aims at finding bugs in software before release, as buggy software is responsible for most security vulnerabilities. Though some researchers in the software security domain may oppose the introduction of a metric to measure security, Voas et al. believe that there should be some way to measure the security of a program before it is released as this will aid in the development of programs with better security. This opposition to the usage of metrics to quantify security is analogous to the opposition to the use of numbers to quantify other quality attributes of software. Voas et al. believe that the one thing that is intrinsically wrong with numbers is that their meanings are often over-sold and caution against the usage of such numbers as absolute indicators of Security.

4.1.2 Extended Propagation Analysis (EPA)

The Adaptive Vulnerability Analysis (AVA) is a dynamic software analysis algorithm that owes its existence to a technique called Extended Propagation Analysis (EPA) (Voas & Miller, 1995; Voas & Miller, 1994) used in assessing safety-critical software.
EPA derives a dynamic failure tolerance metric using Source-Code instrumentation and fault-injection. It was originally designed in order to meet the demands of customers oriented toward safety-critical systems like health care systems. The Algorithm has proven to be successful in assessing Software Safety. This technique is primarily based on fault-injection where a software is dynamically analyzed over multiple runs. The technique collects dynamic information concerning which output variables are affected by a data state value that is somehow altered (Voas & Miller, 1995). EPA uses simulated infections to mimic both programmer faults and hardware failures. A simulated infection is a modified value forced into the value of variable in a data state.

4.1.2.1 EPA Algorithm

Let $S$ denote a specification, $P$ denote an implementation of $S$, $x$ denote a program input, $\Delta$ denote the set of all possible inputs to $P$, $Q$ denote the probability distribution of $\Delta$, $l$ denote a program location in $P$, and let $i$ denote a particular execution of location $l$ caused by input $x$. Let $A_{ipx}$ represent the data state produced after executing location $l$ on the $i^{th}$ execution from input $x$. Let

$$A_{ipx} = \{A_{ipx} \mid 1 \leq i \leq n_x\}$$

and

$$\alpha_{ipx} = \{A_{ipx} \mid x \in \Delta\}$$

where $n_x$ is the number of times location $l$ is executed by input $x$.

Let PRED denote a predication expression that relates specific variables to value ranges or combinations of variables and ranges. This PRED denotes whether a particular undesirable output event occurred.

1. Set count to 0

2. Randomly select an input $x$ according to $Q$ and if $P$ halts on $x$ in a fixed period of time find the corresponding $A_{ipx}$ in $\alpha_{ipx}$. Set $Z$ to $A_{ipx}$.
3. Alter the sampled value of variable \(a\) found in \(Z\), creating \(\tilde{Z}\) and execute the succeeding code on \(\tilde{Z}\). If \(l\) is executed more than once for \(x\), alter \(a\) in each \(A_{ipx.2} \leq i \leq m\).

4. If the output satisfies PRED, increment count

5. Repeat steps 2–4 \(n\) times.

6. Divide count by \(n\) yielding \(\psi_{alPQ};l = \psi_{alPQ}\) is the degree of fault tolerance.

Voas and Miller discuss how this technique can be used with the inverse input distribution \(\overline{Q}\) to mimic rare but legal events (Voas & Miller, 1995). This has the potential to further improve the accuracy of the assessment as it also takes into account inputs that are not normally expected but still have a non-zero probability of occurrence.

Mean-Time-To-Hazard is calculated as

\[
MTTH = \left[ \frac{1}{M} \sum_{l=1}^{M} \left( \frac{\text{number of prog. executions}}{\text{unit of time}} \right) \right]^{-1}
\]

And Minimum Time to Hazard

\[
\text{MinTTH} = \left[ \max_{l} \left( \psi_{alPQ} \right) \left( \frac{\text{number of prog. executions}}{\text{unit of time}} \right) \right]^{-1}
\]

These formulae can be used with both \(Q\) and \(\overline{Q}\) or more realistically with a combination of these.

Voas et al. discuss how the technique was successfully applied to measure the failure tolerance of a safety critical medical application called Magneto Stereotaxis System (MSS). The Value of the metric is a number in the range of 0.0 to 1.0, where 0.0 implies no failure-tolerance and 1.0 represents complete failure-tolerance.
4.2 Adaptive Vulnerability Analysis

Voas et al. have worked to adapt EPA for use in deriving Security Metrics primarily because Software Security is a field with a dearth of dynamic assurance methods and metrics. The first step in this adaptation is the enhancement of EPA’s primitive means for simulating code weaknesses via fault injection mechanisms. AVA uses “perturbation functions” like EPA to force simulated infections into program states. Voas et. al. use a perturbation function flipBit that allows the user to flip any bit from 0 to 1 or vice versa. For test case generation they use both $Q$, the normal operational profile and $\bar{Q}$ the inverse operational profile. In fact the use of the inverse operational profile is especially necessary because sampling from the rare input space may prove to be useful in detecting security vulnerabilities. This can be very useful in the prediction of the software security when the software encounters unusual operational events. Intrusions are specified as predicates.

4.2.1 Algorithm

Let $P$ denote the program, $x$ denote a program input value, $Q$ denote the normal usage probability distribution and $\bar{Q}$ denote the inverse usage probability distribution, and $l$ denote a program location in $P$.

1. For each location $l$ in $P$ that is appropriate, perform steps 2-7.
2. Set count to 0.
3. Randomly select an input $x$ from $Q$ or $\bar{Q}$ and if $P$ halts on $x$ in a fixed period of time find the corresponding data state created by $x$ immediately after the execution of $l$. Call this data state $Z$.
4. Alter the sampled value of variable $a$ found in $Z$ creating $\tilde{Z}$ and execute the succeeding code on $\tilde{Z}$. The manner by which $a$ is altered will be representative of the threat class $T$ that is desired.
5. If the output from P satisfies PRED, increment count.

6. Repeat steps 3-5 n times, where n is the number of input test cases.

7. Divide count by n yielding $\hat{\psi}_{alpq}$, a vulnerability assessment for each line l.

   This means $1 - \hat{\psi}_{alpq}$ is the security assessment that was observed given P, Q and T.

The major challenge with the application of this algorithm is the definition of the threat class T. According to Voas et al. this set includes a set of default perturbation functions representative of known threats and application specific threat classes defined by the user. But because future threats may be so novel, the set of default perturbation functions includes functions that are representative not only of known threats but also random corruptions in the state during program execution. It is possible that some future unanticipated threat may yield a data state very similar to the data state yielded by these random corruptions. Voas. et.al. quote from Daran and Thevenod (Daran & Thevenod, 1995) that the behavior of simple simulated faults nicely mirrors the behavior of actual complex fault classes.

4.2.2 Metrics from AVA

Voas et al. propose the following metrics based on AVA but with a caveat – these metrics are not absolute like mean-time-to-failure but relative allowing users to compare 2 versions of the same system or 2 dissimilar systems providing the same functionality.

4.2.2.1 Minimum-Time-To-Intrude (MinTTI)

This metric can be formalized for any unit of time that is desired and larger the assessed value more secure the system is predicted to be.
\[ MinTTI = \left[ \max_i \left( \hat{\psi}_{alPQ} \hat{\epsilon}_{IPQ} \right) \left( \frac{\text{programexecutions}}{\text{unitoftime}} \right) \right]^{-1} \]

Here \( \hat{\epsilon}_{IPQ} \), the execution probability of location l in program P, is simply the probability that a randomly selected input x selected according to Q will execute location l. The MinTTI measure is based on the location in the program that showed greatest weakness. \( \hat{\psi}_{alPQ} \) does not account for the frequency with which a particular location is executed according to Q and therefore to account for this the \( \hat{\epsilon}_{IPQ} \) measure is included in the equation.

### 4.2.2.2 Mean-Time-To-Intrusion

This is defined as the average time interval before an intrusion will occur based on 3 things: input cases in Q and it’s inverse, the classes of fault injections used and the classes of intrusions defined in PRED.

\[ MTTI = \left[ \sum_{i=1}^{M} \left( \hat{\psi}_{alPQ} \hat{\epsilon}_{IPQ} \right) \left( \frac{\text{programexecutions}}{\text{unitoftime}} \right) \right]^{-1} \]

where M is the number of locations where AVA was applied.

### 4.3 Mutation Testing

Another area of research that has captured the attention of many researchers in the domain of Software Testing is Mutation Testing. It is an approach to test automation that aims to produce test cases that are good at distinguishing between some description N and variants of it (Clark, Haitao, & Robert, 2010). Each variant called a “mutant” is produced by applying a mutation operator. A test case t is said to kill mutant M of N if M and N produce different outputs on execution with t. A mutant M of N is said to be an equivalent mutant if no test case kills M.
The idea is that a test case that is good at distinguishing N from variants of N is likely to be good at finding faults similar to applications of mutant operators.

Mutants are produced by the application of mutation operators. Usually these operators include the replacement of a + by a -, or a < by a <= or a variable by a constant or removal of a sub-expression. The reasoning behind this is the “Competent Programmer Hypothesis” which states that competent programmers make very small mistakes.

Under “strong” mutation testing, a mutant is killed by a test case t only if it produces a different output from the original program when executed with t. In weak mutation testing, a mutant is killed by a test case t if it produces a different value for some variable in the program state immediately following the point of application of the mutation operator. In Firm-Mutation testing the tester is allowed to choose a location l where the original and mutant are supposed to produce different value for some state variable.

The biggest challenge in mutation testing is that the number of mutants produced can be so huge even with the application of a single mutation operator. For this reason, first-order mutants are alone considered – those produced by single application of one mutation operator. The reasoning behind this is the “Coupling Hypothesis” which states that a test case that kills first-order mutants will also generally kill all higher-order mutants.

There is an enormous literature on mutation testing and many results of applying them to programs have been reported (Andrews et. al., 2005; Baudry et. al., 2005). Recently there have also been attempts on applying mutation testing to formal specifications (Black, Okun, & Yesha, 2000; Sugeta, Maldonado, & Wong, 2004; Zhan & Clark, 2005). Here test cases are produced that kill all the mutants of the
specification and these test cases are then applied to the program. This requires a formal language for specification.

4.3.1 Advantages of Mutation Testing

Clark, Haitao, and Robert state a number of advantages of mutation testing (Clark, Haitao, & Robert, 2010).

- It allows the tester to target particular classes of faults. If a program passes a test suite that kills all the non-equivalent mutants, it becomes evident that the non-equivalent mutants produced were not correct. A greater confidence is obtained from a test suite that distinguishes the correct program from the wrong ones.

- Other test criteria may be simulated using mutation testing. If the mutation operator replaces a statement by a new statement that terminates execution with an error message, a test suite that kills all the non-equivalent mutants will provide 100% statement coverage.

4.3.2 Disadvantages of Mutation Testing

The enormous number of mutants produced poses a great overhead to the tester. Clark, Haitao, and Robert report a case where 951 mutants were produced by applying 22 mutation operators to a program containing just 28 executable statements (Clark, Haitao, & Robert, 2010). A workaround for this problem is the “selective mutation” in which a subset of the mutation operators is applied. The presence of equivalent mutants increases the cost of mutation testing and there have also been attempts to preventing the introduction of equivalent mutants and automatic detection of equivalent mutants.
4.3.3 Semantic Mutation Testing

The Mutation Testing described so far is often characterized as syntactic mutation testing as changes are made to the syntax of the program being tested. An alternative is the Semantic Mutation Testing proposed by Clark, Haitao, and Robert (Clark, Haitao, & Robert, 2010). Syntactic Mutation Testing assumes that competent programmer make small mistakes. But these small mistakes may not be always syntactic. Sometimes semantically small mistakes may not be reflected by small syntactic changes. Semantic Mutation Testing modifies the semantics of the language used to specify the description. In Semantic Mutation Testing, the semantic mutants to be produced are highly context sensitive. Clark, Haitao, and Robert illustrate the approach with examples of C Code (Clark, Haitao, & Robert, 2010).

4.3.4 Mutation Testing for Security

Given the ability of Mutation Testing to uncover flaws in the program the question naturally arises if it can reveal Security Flaws. The work of Shahriar is one such attempt (Shahriar, 2008).

4.3.4.1 Mutation Testing to uncover Buffer Overflow Vulnerabilities

Shahriar applies Mutation Testing to uncover buffer overflow vulnerabilities in the ANSI C Language (Shahriar, 2008). The language has been chosen given its widespread use in developing many critical software applications like ftp servers and web servers. Most of the existing mutation operators proposed for C do not take into account the buffer overflow vulnerability in the ANSI C Libraries. The proposed mutation operators are shown in the Table 4.1.
Table 4.1 - Mutation Operators for Buffer Flow Vulnerability (Taken from (Shahriar, 2008))

<table>
<thead>
<tr>
<th>Category</th>
<th>Operator</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutating Library</td>
<td>S2UCP</td>
<td>Replace strncpy with strcpy</td>
</tr>
<tr>
<td>Function Calls</td>
<td>S2UCT</td>
<td>Replace strncat with strcat</td>
</tr>
<tr>
<td></td>
<td>S2UGT</td>
<td>Replace fgets with gets</td>
</tr>
<tr>
<td></td>
<td>S2USN</td>
<td>Replace snprintf with sprintf</td>
</tr>
<tr>
<td></td>
<td>S2UVS</td>
<td>Replace vsnprintf with vsprintf</td>
</tr>
<tr>
<td>Mutating Buffer Size</td>
<td>RSSBO</td>
<td>Replace buffer size with destination buffer size plus 1</td>
</tr>
<tr>
<td>Arguments</td>
<td>RFSNS</td>
<td>Replace “%ns” with “%s”</td>
</tr>
<tr>
<td></td>
<td>RFSBO</td>
<td>Replace “%ns” with “%ms” where m=size of destination buffer plus 1</td>
</tr>
<tr>
<td></td>
<td>RFSBD</td>
<td>Replace “%ns” with “%ms” where m=size of destination buffer plus ∆</td>
</tr>
<tr>
<td></td>
<td>RFSIFS</td>
<td>Replace “%s” with “%ns” where n is the size of the destination buffer</td>
</tr>
<tr>
<td>Mutating Format Strings</td>
<td>MBSBO</td>
<td>Increase buffer size by 1 byte</td>
</tr>
<tr>
<td>Mutating Buffer Variable Sizes</td>
<td>RMNLS</td>
<td>Remove null character assignment statement</td>
</tr>
</tbody>
</table>

For brevity, the killing criteria for each operator have not been specified. There are basically 2 killing criteria $C_1$ and $C_2$. $C_1$ is expressed as $ES_p \neq ES_M$ and $C_2$ is expressed as $Len(Buf_p) \leq N \&\& Len(Buf_M) > N$ where $P$ is the original
implementation, M is the mutant, $ES_P$ is the exit status of P, $ES_M$ is the exit status of M, $Len(Buf_P)$ is the length of buffer in P and $Len(Buf_M)$ is the length of buffer in M. Firm-Mutation testing is used. Shahriar also shows various examples of application of these operators and their results (Shahriar, 2008).

The question that naturally arises is how these operators can uncover buffer overflow vulnerabilities.

- The corruption of return addresses can be detected through test cases that kill mutants generated by S2UCP, S2UCT, S2UGT, S2USN, S2UVS, RFSNS, RFSBD, RFSIFS, and MBSBO operators.
- Attacks that expose BOF by one byte can be detected by test cases that kill mutants generated by RFSBO and RSSBO operators.
- The RMNLS operator helps revealing BOF vulnerabilities that cause arbitrary reading of neighboring variables of buffer in either stack or heap area.

In order to evaluate the proposed operators Shahriar selects 4 benchmark programs that have BOF vulnerabilities (Shahriar, 2008). For each program 2 versions are used – a “bad” one with the vulnerability and a “good” one without the vulnerability. A “good” program may be a patched one or an upgraded version.
The evaluation of the effectiveness of the proposed operators is done in 2 stages. In the first stage, a good and corresponding bad program is obtained and operators are applied to the bad program to generate mutants. An initial test data set is constructed and is gradually increased to bring the Mutation Score MS which is the percentage of the total number of mutants killed to the total number of non-equivalent mutants to 100%. In the second stage, for each of the test data set constructed above it is examined if at least one test case can distinguish between the good program and the corresponding bad program. The results obtained by Shahriar are tabulated in Table 4.3 (Shahriar, 2008).
Table 4.3: Results of applying mutation operators (from (Shahriar, 2008))

<table>
<thead>
<tr>
<th>Name</th>
<th>Total number of mutants</th>
<th>Average MS (%) of bad programs</th>
<th>Average test data size</th>
<th>Percentage of test data sets that reveal buffer overflow vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wu-ftpd</td>
<td>1</td>
<td>100</td>
<td>12.53</td>
<td>100</td>
</tr>
<tr>
<td>Edbrowse</td>
<td>4</td>
<td>100</td>
<td>12.86</td>
<td>100</td>
</tr>
<tr>
<td>Rhapsody IRC</td>
<td>20</td>
<td>100</td>
<td>20.40</td>
<td>100</td>
</tr>
<tr>
<td>Cmdftp</td>
<td>2</td>
<td>100</td>
<td>12.93</td>
<td>100</td>
</tr>
</tbody>
</table>

4.3.4.2 Mutation Testing to uncover SQL Injection Vulnerabilities

Shahriar also applies Mutation Testing to uncover SQL Injection Vulnerabilities (Shahriar, 2008). The methodology adopted is very similar to the one adopted for buffer overflow vulnerabilities. For brevity, only the operators proposed are tabulated in Table 4.4.
Table 4.4 - Mutation Operators for SQL Injection Vulnerability (Taken from (Shahriar, 2008))

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WC (Where Conditions)</td>
<td>RMWH</td>
<td>Remove where keywords and conditions</td>
</tr>
<tr>
<td></td>
<td>NEGC</td>
<td>Negate each of the unit expression inside where conditions</td>
</tr>
<tr>
<td></td>
<td>FADP</td>
<td>Prepend “FALSE AND” after the where keyword</td>
</tr>
<tr>
<td></td>
<td>UNPR</td>
<td>Unbalance parentheses of where condition expressions</td>
</tr>
<tr>
<td>AMC (API method calls)</td>
<td>MQFT</td>
<td>Set multiple query execution flags to true</td>
</tr>
<tr>
<td></td>
<td>OVCR</td>
<td>Override commit and rollback options</td>
</tr>
<tr>
<td></td>
<td>SMRZ</td>
<td>Set the maximum number of records returned in a result set to infinity</td>
</tr>
<tr>
<td></td>
<td>SQDZ</td>
<td>Set query execution delay to infinity</td>
</tr>
<tr>
<td></td>
<td>OVEP</td>
<td>Override the escape character processing flags</td>
</tr>
</tbody>
</table>

4.4 Enhancing AVA with Mutation Testing

To explore the possibility of enhancing the original AVA algorithm with mutation testing, the following change is proposed to step 3 of the AVA algorithm.

3. Select an input x from test cases generated using mutation testing and if P halts on x in a fixed period of time, find the corresponding data state created by x immediately after the execution of l. Call this data state Z.
4.4.1 Case Study

To study the improvement in performance obtained by the proposed modification to AVA, 4 open source programs were investigated and both the AVA and the enhanced AVA were applied. For the generation of test cases required by step 3 of the modified algorithm, the same procedure outlined by Shahriar is followed (Shahriar, 2008). This test data set kills all the generated mutants obtained by applying the operators proposed by Shahriar for detection of buffer overflow vulnerabilities (Shahriar, 2008).

For each of the 4 programs there are 2 versions – the version with the vulnerability and the version with the vulnerability patched. The four open source programs selected are essentially the same as selected by Shahriar to demonstrate the effectiveness of the proposed mutation operators.

4.4.2 Prototype Tool Implementation

A tool that accepts a C program unit was implemented. The location of the C program is specified in the appropriate Text box and a text file that contains all the test cases is created and its location specified in the appropriate text box. On Clicking the “Generate Metrics Using AVA” button the results of applying AVA to the program are displayed and on clicking the “Generate Metrics Using Enhanced AVA” button the results of applying the enhanced version of AVA to the program are displayed. The Tool is developed using VB.NET, .NET Framework 2.0 on Windows XP.
4.4.3 Results and Analysis

The results of applying the AVA and AVA with the proposed enhancement to each of the 4 programs described above using the tool developed for the purpose are tabulated in Tables 4.5 and 4.6.

**Table 4.5 – Results For the Unpatched (Vulnerable) Version**

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Using AVA</th>
<th>Using AVA With the Proposed Enhancement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wu-ftpd-2.6.2</td>
<td>MTI: 0.312 MinTI: 0.305</td>
<td>MTI: 0.293 MinTI: 0.281</td>
</tr>
<tr>
<td>Edbrowse-2.2.10</td>
<td>MTI: 0.298 MinTI: 0.243</td>
<td>MTI: 0.223 MinTI: 0.212</td>
</tr>
<tr>
<td>Rhapsody IRC-0.28b</td>
<td>MTI: 0.372 MinTI: 0.363</td>
<td>MTI: 0.321 MinTI: 0.313</td>
</tr>
<tr>
<td>Cmdftp-0.64</td>
<td>MTI: 0.341 MinTI: 0.332</td>
<td>MTI: 0.297 MinTI: 0.263</td>
</tr>
</tbody>
</table>
Figure 4.2: Results for Unpatched Applications – Enhanced AVA produces low values

Table 4.6 – Results For the Patched Version

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Using AVA</th>
<th>Using AVA With the Proposed Enhancement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MTTI</td>
<td>MinTTI</td>
</tr>
<tr>
<td>Wu-ftpd-2.6.2</td>
<td>0.778</td>
<td>0.732</td>
</tr>
<tr>
<td>Edbrowse-2.2.10</td>
<td>0.812</td>
<td>0.792</td>
</tr>
<tr>
<td>Rhapsody IRC-0.28b</td>
<td>0.761</td>
<td>0.753</td>
</tr>
<tr>
<td>Cmdftp-0.64</td>
<td>0.897</td>
<td>0.803</td>
</tr>
</tbody>
</table>
As can be observed from the results, the AVA with the proposed enhancement tends to give more accurate estimates of software security as is evinced by the low values it reports for the unpatched version compared to the standard AVA. A lower value for MTTI (and MinTTI) indicates a compromised security state. On the other hand, for the patched version the difference between the 2 tends to narrow down. Because the vulnerability under consideration has been patched, the attack surface is narrowed down and hence the higher values for MTTI and MinTTI.

4.5 Conclusion

Assessment of Software Security has become pivotal in the current era and attempts to provide quantitative measures of software security should prove very useful. Metrics can be very useful in assessing security provided they are used as relative measures and not absolute ones. AVA provides us with exactly such metrics. Mutation testing has the potential to uncover security vulnerabilities in software. Thus the combination of AVA with mutation testing yields better results as expected.