inconvenience (e.g., not requiring users to be re-trained for every change of state), efficient in time and resources (e.g., minimum delay and disruption), etc.

More specifically, usability reflects the tradeoffs among ease of use, performance, and security.

Definition 5. Usability complements the other evaluation metrics by defining how easy of use the MTD system is. An ideal MTD system should maximize ease of use and minimize the performance penalty while achieving maximally possible security.

Note that we do not consider hardware/software cost when comparing MTD approaches, because the comparable MTD approaches are from the same category and hence have similar hardware/software requirements.

4.3. Aggregation of Evaluation Metrics by Analytic Hierarchy Process (AHP)

To achieve a comprehensive and systematic evaluation, we need fully consider different types of information, which each relates to a specific criterion/attribute. Hence, after identifying the criteria for evaluation, we further adopt a methodology to aggregate them and provide a comprehensive analysis. Many analytical tools have been discussed to address these problems associated with the field of decision analysis, such as Multiple Attribute Utility Theory (MAUT) [27], Multiple-Attribute Decision Analysis (MADA) [28], Multiple Correspondence Analysis (MCA) [29] and Analytic Hierarchy Process (AHP) [30], to name a few.

We adopt the multi-criteria evaluation methodology named Analytic Hierarchy Process (AHP), for measuring the relative strength of MTD approaches. AHP [30, 31] plays an important role in many real world decision situations such as government, business, industry, healthcare and education. It provides the decision makers a comprehensive and rational framework for structuring a decision problem, for representing and quantifying its elements, for relating those elements to overall goals and their understanding of the problem, and for evaluating alternative solutions (A running example on how to choose a company leader can be found in [32]).

AHP suits our problem setting very well because of the following reason. For MTD approaches of different categories (e.g., software diversity, address space/data/instruction set randomization, N-version), their actual security and performance concerns vary a lot, because the type and size of attack surface, the likelihood of successful attacks, as well as cost of dynamically changing attack surface in each category are very different from one another. As such, although the five metrics we propose are generic, their relative weights in final evaluation would vary for different categories. Hence, to determine the best approach in each category, we will first understand the network/system model, the security model and the cost model for each category. We can leverage AHP to evaluate the alternative MTD approaches in each category against each criterion/metric that measures how well a method accomplishes a particular criterion. Then we compare the alternative MTD approaches by generating a score of each alternative MTD approach for ranking. Note that a very attractive feature of AHP is that for comparison purpose it will help generate relative scores for criteria which cannot be directly quantified with an absolute meaning.

General Principles of AHP. Specifically, using AHP we can first construct a hierarchy, as shown in Figure 4. This hierarchy has three levels: the top one is our goal to find the best MTD strategy in a category, the second one includes the five criteria we proposed, and the bottom one includes the alternative MTD approaches to evaluate. Once the hierarchy is built, we can systematically evaluate its various elements by comparing them in a pairwise way, with respect to their impacts on an element above them in the hierarchy.

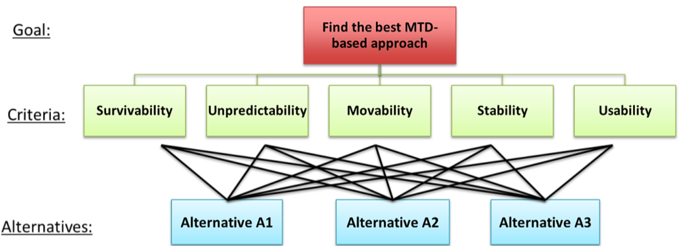

Figure 4. An AHP hierarchy to choose the best MTD approach

For example, we will start from the bottom level by comparing each pair of alternatives w.r.t. each of the five metrics above and totally we will perform $3 * 5 = 15$ comparisons. During the comparison, we calculate numerical weights (priorities) for each of the decision alternatives and these numbers represent the alternatives' relative ability to achieve the decision goal. For each criterion (metric), the weights of all alternatives are then transferred to an AHP matrix to calculate the priority of each alternative. After evaluating the alternatives with respect to their strength in meeting the criteria, we will then evaluate the criteria with respect to their importance in reaching the overall goal. Following a similar process, each criterion will be given a weight w.r.t. the goal. Finally, with all the priorities of the criteria with respect to the goal, and the priorities of the alternatives with respect to the criteria, we can synthesize and calculate the priorities of the alternatives with respect to the goal. The one with the highest priority will be the winner. (we will show a concrete example later in the case study of Section 5.)

A Detailed Procedure of AHP. Suppose that m criteria are considered and n alternatives are to be evaluated. In our case, $m = 5$ and $n = 3$. The AHP can be implemented in three consecutive steps [33, 34].

Step 1. Computing the vector of criteria weights: The AHP starts with creating a pairwise comparison matrix A. The matrix A is a $m \times m$ matrix with real numbers. Each entry a_{jk} in matrix A represents the importance of jth criterion relative to the kth criterion. The entries a_{jk} and a_{kj} satisfy the following constraint: $a_{jk} \times a_{kj} = 1$. Obviously, $a_{jj} = 1$ for all $1 \leq j \leq m$. The relative importance between two criteria is measured according to a numerical scale from 1 to 9 [34].

Once the matrix A is constructed, it is possible to derive from A the normalized pairwise comparison matrix A_{norm} by summing all the entries in each column, then each entry $\overline{a}_{jk}$ of the matrix A_{norm} is computed as $\overline{a}_{jk} = \frac{a_{jk}}{\sum_{l=1}^{m} a_{lk}}$, i.e.,

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix} \xrightarrow{\overline{a}_{jk} = \frac{a_{jk}}{\sum_{l=1}^{m} a_{lk}}}$$

$$A_{norm} = \begin{pmatrix} \overline{a}_{11} & \overline{a}_{12} & \overline{a}_{13} & \overline{a}_{14} & \overline{a}_{15} \\ \overline{a}_{21} & \overline{a}_{22} & \overline{a}_{23} & \overline{a}_{24} & \overline{a}_{25} \\ \overline{a}_{31} & \overline{a}_{32} & \overline{a}_{33} & \overline{a}_{34} & \overline{a}_{35} \\ \overline{a}_{41} & \overline{a}_{42} & \overline{a}_{43} & \overline{a}_{44} & \overline{a}_{45} \\ \overline{a}_{51} & \overline{a}_{52} & \overline{a}_{53} & \overline{a}_{54} & \overline{a}_{55} \end{pmatrix}$$

Finally, the criteria weight vector w (that is an m-dimensional column vector) is built by averaging the entries on each row of A_{norm}, i.e.,

$$w_j = \frac{\sum_{l=1}^{m} \overline{a}_{jl}}{m}, \text{ for } w = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{pmatrix}$$

Step 2. Computing the matrix of alternative scores: The matrix of alternative scores is a $n \times m$ matrix S with real numbers. Each entry s_{ij} of S represents the score of the ith alternative with respect to the jth criterion. In order to derive such scores, a pairwise comparison matrix $B^{(j)}$ is first built for each of the m criteria. The matrix $B^{(j)}$ is a $n \times n$ matrix with real values. Each entry b_{ih} of the matrix $B^{(j)}$ represents the evaluation of the ith alternative compared to the hth alternative with respect to the jth criterion. Similarly, the entries b_{ih} and b_{hi} satisfy the following constraints: $b_{ih} \times b_{hi} = 1$ and $b_{ii} = 1$ for all i. An evaluation scale [34] will be used to translate the decision maker's pairwise evaluations into numbers.

Second, the AHP applies to each matrix $B^{(j)}$ the same two-step procedure described for the pairwise comparison matrix A, i.e., it divides each entry by the sum of the entries in the same column, and then it averages the entries on each row, thus obtaining the score vectors $s^{(j)}, j = 1, \cdots, m$. The vector $s^{(j)}$ contains the scores of the evaluated alternatives with respect to the jth criterion.

Finally, the score matrix S is obtained as $S = [s^{(1)} \cdots s^{(m)}]$, i.e., the jth column of S corresponds to $s^{(j)}$.

Step 3. Ranking the alternatives: Once the weight vector w and the score matrix S have been computed, the AHP obtains a vector v of global scores by multiplying S and w, i.e., $v = S \times w$. The ith entry v_i of v represents the global score assigned by the AHP to the ith alternative.

AHP incorporates an effective technique for checking the consistency of the evaluations made by the decision maker when building the pairwise comparison matrices involved in the process, namely the matrix A and the matrices $B^{(j)}$. The technique relies on the computation of a suitable consistency index CI. CI is obtained by first computing the scalar λ as the average of the elements of the vector whose jth elements is the ratio of the jth element of the vector $A \times w$ to the corresponding element of the vector w. Then, $CI = \frac{\lambda - m}{m-1}$. A perfectly consistent decision maker should always obtain $CI = 0$, but small values of inconsistency could be tolerated.

5. A Case Study on Software Diversification MTD

To demonstrate the applicability of our proposed evaluation framework, next we present a case study on a network level approach - dynamic software diversity based MTD.

5.1. Software Diversification MTD

Software diversity [7–9], in spirit of survivability through heterogeneity, has been one of the major MTD approaches. Specifically, the purpose of software diversity is to select and deploy a set of off-the-shelf software to hosts in a networked system, such that the number and types of vulnerabilities presented on one host would be different from that on its neighboring nodes. In this way, one would be able to contain an automated worm attack in an isolated "island".

An illustrating example is showed in Figure 5. We use an undirected graph as the abstraction of a general networked system. Example networked systems include intranet, enterprise social networks, tactical mobile ad hoc networks, and wireless sensor networks of different network topologies. In this figure, there are 11 machines represented by nodes and 5 distinct pieces of vulnerable software represented by different colors. An attack can propagate by exploring one type of vulnerability (color). From the figure, we can see that a successful attack exploiting the green color can compromise up to four machines ($v2, v5, v7, v11$), but it can only compromise one machine when it exploits the yellow color as machines with the yellow color cannot communicate directly.

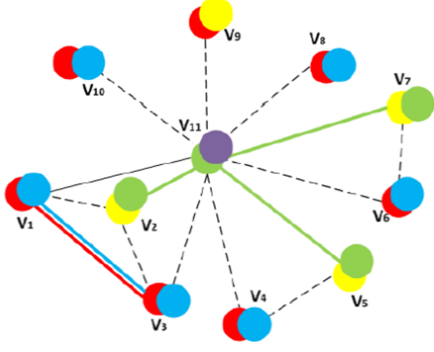

Figure 5. Network topology utilizing a diverse software distribution. Dashed or solid lines mean two nodes can communicate directly (e.g., through TCP/IP or being friends in a social network). Solid lines further indicate two nodes share at least one common color.

5.2. Quantifying Five Evaluation Metrics

Next, we discuss how to instantiate and quantify five general evaluation metrics.

Survivability. According to Figure 5, a defective edge between two nodes (which share the same color) indicates that the exploitation of one type of vulnerability on one host can lead to the compromise of the other. The size of a connected component indicates the number of compromised machines if the corresponding vulnerability is discovered and exploited by the attacker (e.g., via a worm attack). We denote a connected component as a common vulnerability graph (CVG). If one can effectively limit the size of the largest CVG, system survivability can then be improved. A better software assignment algorithm should be able to produce a software assignment solution with a smaller largest CVG. Formally, the survivability of a networked system can be computed as follows:

$$Survivability = 1 - s_{max}(c_i)/N, \quad (1)$$

where $s_{max}(c_i)$ denotes the size of the largest CVG that is formed by color c_i, and N denotes the network size.

Unpredictability. The goal of the attacker is to compromise as many nodes as possible; thus, the vulnerability (color) of the largest CVG is always the attackerŠs first choice to exploit. Unpredictability of a software assignment strategy in this case describes the difficulty for an attacker to determine the prevailing color (which forms the largest CVG) after shuffling. For example, given two software assignment algorithms, to compare their unpredictability, we can observe the distribution of colors for the largest CVGs across a number of software shufflings. If the prevailing colors are uniformly randomly distributed, it would be hard for an attacker to learn the pattern and predict the next prevailing color. The attacker has to try out every color with about an equal probability in order to compromise the network to the largest extent.

To formulate unpredictability of software assignment in a quantitative way, we use entropy to measure the expected or average ŚsurpriseŠ over all shufflings, reflecting the uncertainty of the prevailing color before it is determined. If the color of the largest CVG is c, let $p(c_i)$ be the probability $c = c_i$. Given a set of colors C and the probabilities of their occurrences, unpredictability produced by the software diversity algorithm is quantified as:

$$Unpredicability = \sum_{c_i \in C} -p(c_i) \times \ln(p(c_i)). \quad (2)$$

Movability. In order to survive from long-lasting (persistent) attacks, software diversity mechanisms should further adopt the technique of software shuffling. By (periodically) re-allocating software on the machines, the attack surfaces of the systems continually change to confuse the attacker and thus delay the attack. In practice, however, to make the assignment solution generated after each shuffling acceptable, the software assignment algorithm needs to take a number of realistic constraints into account. A software assignment algorithm may be able to well or only partially accommodate the practical constraints that give rise by host and software requirements. Here host constraint means that certain hosts should be installed with some specific types of software to perform required functionality (e.g., to deploy a database server it is required to assign DB2). Software constraint means certain combination of software should (or should not) be assigned to specified hosts simultaneously (e.g., PHP, Apache, MySQL and Linux need to be assigned together to implement LAMP on a single node).

Besides, in practice the constraints are not equally important. Some of the constraints are critical and thus cannot be violated, for example, the case of LAMP - a lack of any one of these four components would cause a service failure on the web server. On the other hand, some constraints are less critical and thus can be relaxed to some extent without impacting the essential functionality of the machine. For example, suppose there is a constraint that a PC should not install a program (for the lack of understanding of its security). If a software assignment algorithm has to assign the program to this machine for greater security of the network, one may, for example, relax this constraint by launching it through a browser-based SaaS cloud service.

We assume that a software assignment algorithm unable to satisfy the practical constraints is less appropriate than an algorithm that well accommodates them. Thus we propose to use penalty score to quantitatively determine the functionality loss caused by violations of given practical constraints. Specifically, we initially assign a penalty score to each constraint

reflecting its significance and the penalty is only applied when that particular constraint is violated. For instance, penalty(x_i) denotes the penalty score of violating constraint x_i. For the most critical constraints that cannot go against, we assign an infinite value to them. In this way, given a set of constraints $CSTR = \{x_1, x_2, \cdots, x_m\}$, the total penalty score for the system is given as:

$$Penalty = \sum_{x_i \in CSTR}^{m} penalty(x_i) \times d(x_i), \qquad (3)$$

where

$$d(x_i) = \begin{cases} 1, & \text{if } x_i \text{ is violated} \\ 0, & \text{otherwise} \end{cases}$$

specifies which constraint is violated.

Stability. Stability of a software assignment algorithm measures variation of the quality of the assignment solutions, in terms of variation of the largest CVG sizes generated by shuffling. In this case, we equate stability with variability. Note that for the unpredictability of shuffling, ideally the shuffling process should be stateless, i.e., each shuffling is independent of the history. Standard deviation can be used to quantify the average difference between assignment solutions, independent of the temporal order in which each assignment solution is generated. The less the variation of the largest CVG size, the more stable of a given software assignment algorithm. Let $E(S)$ be the mean size and N be the number of shufflings, the stability of the software assigning algorithm is quantified as:

$$Stability = \sqrt{\frac{1}{N} \sum_{i=1} (S_i - E(S))^2}. \qquad (4)$$

Usability. The combination of the software installed in a machine is prone to variation due to shuffling. For example, Microsoft Office may be substituted by Open Office and Firefox may be substituted by Internet Explorer or Google Chrome. Among all these options, some software products are easier to operate as compared to others. Besides, users tend to choose software that they are familiar with or have certain preference with. A software assignment algorithm might cause inconvenience to users in accomplishing their tasks (e.g, when a secretary's computer cannot install Microsoft Word despite the fact she has been using it in the past years), and it is very likely not a comfortable assignment for some users. Thus, for software diversity algorithms, usability is defined to reflect user's experience regarding the assigned software, e.g., familiarity, comfort, satisfaction and ease of use from a user's point of view. A good algorithm should be able to take user's concerns as one input and maximize overall usability.

We will use acceptance rate (a real value from 0 to 1) to measure user's satisfaction level, reflecting their attitudes toward shifting from a particular software product to another one. We first categorize software products based on their functionalities. For example, Linux, Snow Leopard, Windows are in the operating system category while Firefox, Chrome, IE and Opera are in the web browser category. Software may be replaced by another one in the same category. Software substitution with high acceptance rate indicates users are satisfied or at least have little trouble with the assignment. Low acceptance rate, on the other hand, indicates that a user finds it inconvenient (or even being prevented from doing his job) switching to new software.

There are two methods to measure usability of a software shuffling. The first one asks users to assign an acceptance rate for every software substitution (in pairwise) in each category based on their experience and attitude. Given these ratings, one can automatically compute the overall acceptance rate for each assignment (compared with the previous assignment) and check whether it is optimal. The other way is to conduct a survey after each shuffling. Users are asked to provide their feedbacks/scores indicating their willingness to accept or reject the assignment of software in their systems. By adding up the scores from users, the final score is then used as the overall usability of a software assignment. Generally speaking, the first approach is more preferable as it only conducts user survey once. A good algorithm may take into consideration individual user acceptance rates when running.

5.3. Evaluation Results

We first evaluate three software diversification algorithms in terms of our general evaluation metrics including survivability, unpredictability, movability, and stability (usability is subject to the user survey results), which builds a foundation for our AHP procedure. Then, we use AHP procedure to produce the best alternative/option among three algorithms for this category. Three software diversification algorithms under our consideration are:

- Algorithm I: basic software diversity algorithm [35];
- Algorithm II: an adjusted software assignment algorithm [36];
- Algorithm III: an Ant Colony Optimization (ACO) based software assignment algorithm [18].

Figure 6 shows an example to compare three software assignment algorithms in terms of survivability. Here the x-axis is the ratio *#weight/#color*, where *#weight* is the number of vulnerable software assigned to a single machine and *#color* is the total number of vulnerable

software installed in the whole network system. This ratio implicitly reflects the likelihood of sharing same colors among nodes. The y-axis is the size of the largest CVG. As we can see, Algorithm I outperforms the other two algorithms by creating smaller s_{max} under the same situation. Algorithm III is better than Algorithm II. Thus it is straightforward to rank these algorithms as Algorithm I > Algorithm III > Algorithm II, in terms of survivability. Note that in practice it is not necessarily always the case one algorithm outperforms the others all the time. (These three algorithms will be used for illustration purpose throughout this paper.)

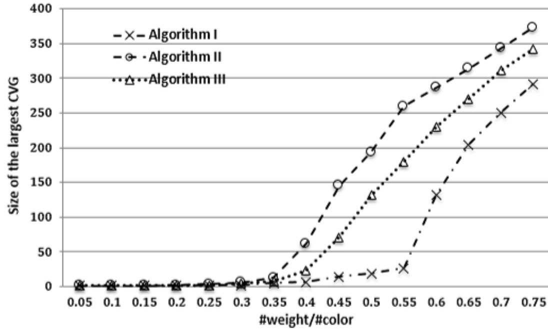

Figure 6. Survivability of different algorithms

To gain intuition for unpredictability, we show below an illustrative example on distribution of the prevailing color (of the largest CVG). As observed in Figure 7, in Algorithm III, $s_{max}(c_i)$ is formed by any color c_i among all the available colors with approximately the equal probability, which indicates the random nature of assignments generated by Algorithm III. According to Shannon theory, when $p(c_i) = p(c_j)$ for any $i \neq j$, unpredictability (entropy) reaches a maximum. As for Algorithm I and Algorithm II, they are more predictable to the attacker than Algorithm III. For example, in the shuffling outcome of Algorithm I, color 1 is more likely to cause largest CVGs, hence more preferable for the attacker to exploit.

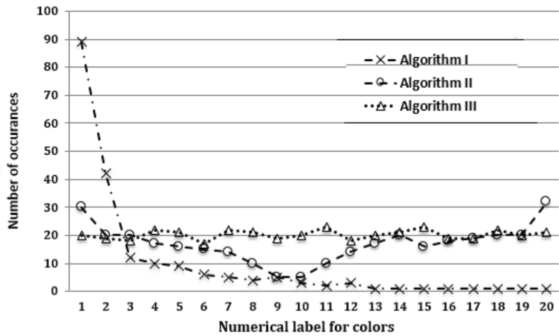

Figure 7. Unpredictability of different algorithms. Here totally 20 colors (vulnerable software) are assigned in the networked system, and the x-axis is the numerical label for each color.

Figure 8 is an example that plots the moving penalties for three algorithms. Each data point in this figure is obtained by calculating the penalty score for a corresponding software assignment. It is observed that, in general, the penalties resulted from Algorithm II are the highest, indicating the performance of Algorithm II is largely restricted by practical constraints (lowest movability). Algorithm III cannot accommodate constraints well either. Algorithm I has the least penalty scores, so it offers better software assignment strategies over Algorithm II and Algorithm III in terms of movability.

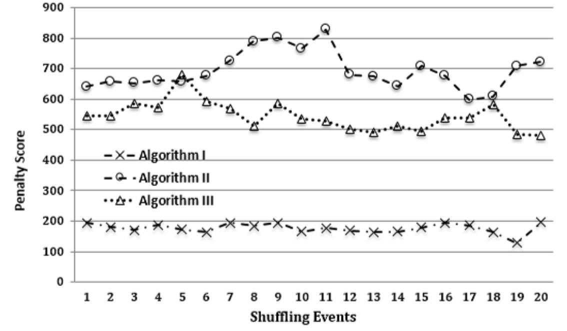

Figure 8. Movability of different algorithms

To fully evaluate the stability of an algorithm, we will need to try different network settings. In this way, one can see the ability of the assigning algorithm to accommodate mutable environments (e.g., different types of network topologies such as scale-free, random or regular graph). Again we use the three algorithms as an example to explain the concept of stability. Suppose we run each of the three algorithms twenty times while changing network topologies and use all generated assignment solutions. In Figure 9, we observe that the variation range of $s_{max}(c_i)$ of Algorithm II is much smaller, which means its standard deviation is lower than the other two. Hence, it produces more stable results, even though the size of $s_{max}(c_i)$ of Algorithm II is larger than Algorithm I (that is, Algorithm II is inferior to Algorithm I in terms of survivability).

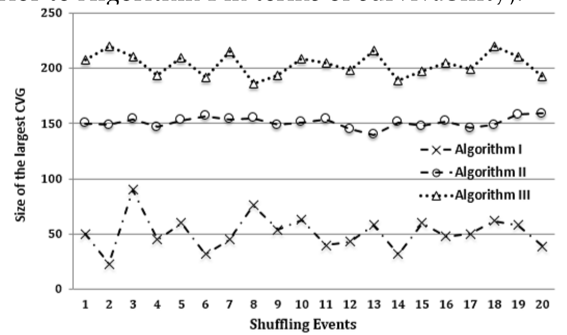

Figure 9. Stability of different algorithms

Table 1 is a simple illustrative example to compare and rank three available software diversity algorithms. We consider the five proposed metrics as the evaluation criteria for ranking (if more metrics are required to be considered, this example can be expanded accordingly). We are interested in comparing relative strength of the alternative software assignment algorithms and determining the best one in terms of the proposed evaluating metrics.

Table 1. Calculate running weights for evaluation metrics (CI=0.8%)

Metrics Ranking	Survivability	Unpredictability	Movability	Stability	Usability	Weights
Survivability	1	1/2	1/2	2	4	0.175
Unpredictability	2	1	2	4	9	0.415
Movability	2	1/2	1	3	6	0.273
Stability	1/2	1/4	1/3	1	2	0.092
Usability	1/4	1/9	1/6	1/2	1	0.045

Table 2. Final scores for every algorithm

Alternatives \Metrics	Survivability (0.175)	Unpredictability (0.415)	Movability (0.273)	Stability (0.092)	Usability (0.045)	Score
Algorithm I	0.571	0.143	0.571	0.143	0.333	0.343
Algorithm II	0.143	0.286	0.143	0.571	0.333	0.25
Algorithms III	0.286	0.571	0.286	0.286	0.333	0.406

First, we need a judgment matrix for determining weights of the 5 metrics according to their importance. In comparing the 5 metrics, for illustration purpose, we assume they are ordered as Unpredictability > Movability > Survivability > Stability > Usability based on their importance. The weight assigned to each metric can then be determined by adopting a scale referred to as 9-point scale of measurement [30]. The following 5×5 matrix contains all of the pairwise comparisons for the metrics, as shown in Table 1.

The matrixes for metrics of the MTDs are omitted here because the value for each metric is mostly given in the previous table. The final decision matrix for this problem can be generated and the final scores are shown in Table 2.

Finally, the ranking for these three software diversity algorithms is: Algorithm III > Algorithm I > Algorithm II, as shown in Figure 10.

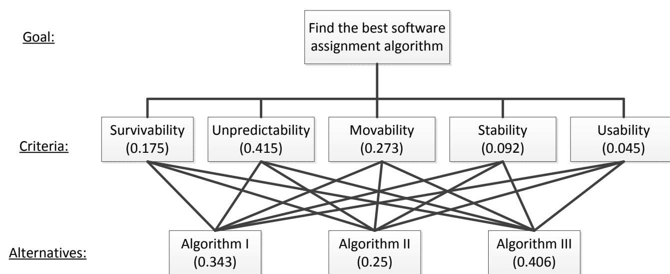

Figure 10. Evaluation result

6. Discussions

In this section, we discuss i) how to apply our generic evaluation framework to other MTD categories, such as runtime-based diversification and network-based diversification; ii) how to apply our generic evaluation framework in different levels, such as system-level, platform-level, and network-level.

6.1. Evaluations on Other MTD Categories

In Section 2 we discussed four different MTD categories and in Section 5 we discussed a detailed case study on a specific MTD category called software diversification. In this Section, we discuss how to apply our generic evaluation framework to other three MTD categories. Since our five evaluation metrics are general, they could be applied to all other three MTD categories. The ways that we instantiate or quantify them might be different for each different category. The idea of Analytic Hierarchy Process (AHP) in the generic evaluation framework is similar, where five general evaluation metrics might have different weights for different categories and the alternative algorithms in each MTD category will be different. Therefore, the flowchart for our generic evaluation framework will stay the same for all the MTD categories evaluations and comparisons and our generic framework will work for all the MTD categories.

6.2. Applying the Proposed Generic Evaluation Framework in Different Levels

Our generic evaluation framework can also be applied in different levels, including system-level, platform-level, and network-level. Software diversification is a network-level solution, so we have already seen the instantiation of our generic framework on network-level. For system-level, we consider a specific operating system on a machine, then all the general evaluation metrics will be defined in this domain. Also, for platform-level, we consider a single machine with potentially multiple operating systems, then our

general evaluation metrics need to be defined for this scope. Other than this, the AHP procedure is similar. So, our generic evaluation framework will apply to the system-level and platform-level, too.

7. Conclusion and Future Work

In this paper, we carefully choose five general metrics for MTD evaluations and comparisons, including survivability, unpredictability, movability, stability, and usability. We also aggregate these five evaluation metrics by for the first time proposing a generic evaluation framework based on Analytic Hierarchy Process (AHP). We work on a detailed case study under a specific MTD category named software diversification with numerical evaluation results, which validates the effectiveness of our generic evaluation framework. Our evaluation framework can be easily ported and applied to other MTD categories (such as runtime-based diversification and network-based diversification) and in different levels. Finally, we discuss the ways to do them.

Our future work includes applying our generic evaluation framework to all other three MTD categories. Besides software diversification, we will study all the other three cases, including runtime-based diversification, network-based diversification, and dynamic platform techniques, in details by instantiating our general evaluation metrics and AHP procedure. We believe that our generic MTD evaluation framework will be effective and efficient for all the MTD categories in different scope/level.

References

[1] Jajodia, S., Ghosh, A.K., Swarup, V., Wang, C. and Wang, X.S. (2011) *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, **54** (Springer).

[2] Giuffrida, C., Kuijsten, A. and Tanenbaum, A. (2012) Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*.

[3] Bojinov, H., Boneh, D., Cannings, R. and Malchev, I. (2011) Address space randomization for mobile devices. In *Proceedings of the fourth ACM conference on Wireless Network Security* (ACM).

[4] Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C. and Sadeghi, A.R. (2013) Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy* (IEEE).

[5] Gundy, M.V. and Chen, H. (2009) Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *NDSS*.

[6] Boyd, S.W., Kc, G.S., Locasto, M.E., Keromytis, A.D. and Prevelakis, V. (2010) On the general applicability of instruction-set randomization. *IEEE Transactions on Dependable and Secure Computing* **7**(3): 255–270.

[7] O'Donnell, A. and Sethu, H. (2004) On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM conference on Computer and communications security* (ACM).

[8] Yang, Y., Zhu, S. and Cao, G. (2008) Improving sensor network immunity under worm attacks: a software diversity approach. In *Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing* (ACM).

[9] Mont, M.C., Baldwin, A., Beres, Y., Harrison, K., Sadler, M. and Shiu, S. (2002) *Towards Diversity of Cots Software Applications: Reducing Risks of Widespread Faults and Attacks*. Tech. Rep. UK HPL-2002-178, HP Laboratories, Bristol.

[10] Jia, Q., Sun, K. and Stavrou, A. (2013) Motag: Moving target defense against internet denial of service attacks. In *22nd International Conference on Computer Communications and Networks (ICCCN)* (IEEE).

[11] Crouse, M. and Fulp, E. (2011) A moving target environment for computer configurations using genetic algorithms. In *4th Symposium on Configuration Analytics and Automation (SAFECONFIG)* (IEEE).

[12] Cui, A. and Stolfo, S. (2011) *Symbiotes and Defensive Mutualism: Moving Target Defense* (in Moving Target Defense, Springer).

[13] Al-shaer, E. (2011) *Toward Network Configuration Randomization for Moving Target Defense* (in Moving Target Defense 2011, Springer).

[14] Wikipedia (2017), Frequency-hopping spread spectrum. URL http://en.wikipedia.org/wiki/FHSS.

[15] Zhuang, R., DeLoach, S.A. and Ou, X. (2014) Towards a theory of moving target defense. In *MTD workshop*.

[16] Zhuang, R., Bardas, A.G., DeLoach, S.A. and Ou, X. (2015) A theory of cyber attacks: A step towards analyzing mtd systems. In *MTD workshop*.

[17] Zhuang, R. (2015) *A Theory for Understanding and Quantifying Moving Target Defense*. Ph.D. thesis, Kansas State University.

[18] Huang, C., Zhu, S. and Guan, Q. (2015) Multi-objective software assignment for active cyber defense. In *CNS*.

[19] Jiang, X., Wang, H.J., Xu, D. and Wang, Y.M. (2007) Randsys: Thwarting code injection attacks with system service interface randomization. In *SRDS*.

[20] Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N. and Boneh, D. (2004) On the effectiveness of address-space randomization. In *CCS*.

[21] Xu, J., Guo, P., Zhao, M., Erbacher, R.F., Zhu, M. and Liu, P. (2014) Comparing different moving target defense techniques. In *MTD workshop*.

[22] Zaffarano, K., Taylor, J. and Hamilton, S. (2015) A quantitative framework for moving target defense effectiveness evaluation. In *MTD workshop*.

[23] Lamb, C. and Hamlet, J. (2016) Dependency graph analysis and moving target defense selection. In *MTD workshop*.

[24] Taylor, J., Zaffarano, K., Koller, B., Bancroft, C. and Syversen, J. (2016) Automated effectiveness evaluation of moving target defenses: Metrics for missions and attacks. In *MTD workshop*.

[25] Prakash, A. and Wellman, M. (2015) Empirical game-theoretic analysis for moving target defense. In *MTD workshop*.
[26] andăHodaăMaleki, S.V., Koch, W., AzerăBestavros and van Dijk, M. (2016) Markov modeling of moving target defense games. In *MTD workshop*.
[27] Kainuma, Y. and Tawara, N. (2006) A multiple attribute utility theory approach to lean and green supply chain management. *International Journal of Production Economics* **101**(1): 99–108.
[28] Yoon, K. and Hwang, C.L. (1995) *Multiple Attribute Decision Making: an Introduction* (Sage).
[29] Abdi, H. and Valentin, D. (2007) *Multiple Correspondence Analysis, Encyclopedia of Measurement and Statistics*.
[30] Saaty, T. (2008) Decision making with the analytic hierarchy process. *International Journal of Services Sciences* **1**(1): 83–98.
[31] Saaty, T. (1990) How to make a decision: the analytic hierarchy process. *European journal of operational research* **48**(1): 9–26.
[32] (2017), AHP Example. URL https://en.wikipedia.org/wiki/Analytic_hierarchy_process.
[33] Saaty, T. (1980) *The Analytic Hierarchy Process* (New York: McGraw-Hill).
[34] (2017), The analytic hierarchy process. URL www.dii.unisi.it/mocenni/Note_AHP.pdf.
[35] Huang, C., Zhu, S. and Erbacher, R. (2014) Toward software diversity in heterogeneous networked systems. In *Proceedings of the 28th IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSec)*.
[36] Huang, C., Zhu, S., Guan, Q. and He, Y. (2017) A software assignment algorithm for minimizing worm damage in networked systems. *Journal of Information Security and Applications* .