Toward Software Diversity in Heterogeneous Networked Systems

Chu Huang[1], Sencun Zhu[1,2], and Robert Erbacher[3]

[1] School of Information Science and Technology, Penn State University
[2] Department of Computer Science and Engineering, Penn State University
[3] U.S. Army Research Laboratory(ARL)

Abstract. When there are either design or implementation flaws, a homogeneous architecture is likely to be disrupted entirely by a single attack (e.g., a worm) that exploits its vulnerability. Following the survivability through heterogeneity philosophy, we present a novel approach to improving survivability of networked systems by adopting the technique of software diversity. Specifically, we design an efficient algorithm 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, we are able to contain a worm in an isolated "island". This algorithm addresses software assignment problem in more complex scenarios by taking into consideration practical constraints, e.g., hosts may have diverse requirements based on different system prerequisites. We evaluate the performance of our algorithm through simulations on both simple and complex system models. The results not only confirm the effectiveness and scalability of our algorithm, but also show its capability in creating moving attack surface. The level of heterogeneity our algorithm can actually create depends on the ratio of the number of installed software to the total number of available software.

1 Introduction

With the fast advancement of nowadays information technology, organizations are becoming ever more dependent on interconnected systems for carrying on everyday tasks. However, the pervasive interdependence of such infrastructure increases the risk of being attacked and thus poses numerous challenges to system security. One major problem for such networked environments is software monoculture [1,2]–running on the risk of exposing a weakness that is common to all of its components, it facilitates the spread of attacks and enables large-scale exploitations that could easily result in overall crash. Considering the consequences of software monoculture in intensively connected systems, there is an urgent need to control the damage of automated attacks that takes advantage of the connectivity of the networked system.

In contrast to homogeneous systems by software monoculture, heterogeneous architectures are expected to have higher survivability [3–5]. This point is very much like the maintenance of genetic and ecosystem diversity in biology. The variability in the biological world allows at least a portion of species to survive an epidemic. Inspired by such phenomena of biodiversity, a good number of techniques have been proposed to improve system resilience and survivability under attacks. However, previous approaches cannot fully meet three highly desired requirements: (R1) *Resistance* against automated attacks in a networked environment; (R2) *Dynamics/moveability* with the environment so that the attack surface can be changed over time; (R3) *Practicability* of the solutions under real-world constraints. To see how existing approaches are limited in meeting these three requirements, we classify them into three main categories: software diversity at the system level, software diversity at network level and N-version programming 1) Diversity at the system level is achieved mainly through randomization techniques, which are limited to individual machines and it is not clear if and how they can be extended to improve the survivability of the networked systems as a whole. 2) Diversification methods at the network level compensate the limitations of the former approach, but it suffers from the problem of only considering single version assignment of software. In the real world scenarios, however, a host (i.e., a commodity PC) typically is required to install with more than one software (i.e., operating system, web browser, email client, office suite applications etc.) to perform particular tasks. 3) N-version programming: achieves higher system survivability depending on its underlying multiple-version software units that tolerate software faults. This method has very high computational cost and is not practical enough to be used routinely in real-world organizations.

In this study we propose a software diversity-based approach to address the problem of survivability in the complex networked systems under automated attacks, with the goal to meet all the design requirements. First, by assigning appropriate software to hosts considering the network connectivity, our approach enhances the system's resistance to automated attacks. Second, given the possibility of accommodating our algorithm into a dynamic environment, our method increases the unpredictability (or movability) of the system. Third, our algorithm considers the real-world constraints on resource allocation, which makes it more practical from implementation point of view. We also demonstrate via simulations that the assignment solution generated by our algorithm is better than previous assigning methods, and very close to the optimal solution. Through experiments we found that the level of heterogeneity in our algorithm depends on the ratio of the number of software installed to the total number of available software, and we also identified critical ratio points for different representative topologies. The capability and possibility of our algorithm in creating moving target defense are also evaluated. Our findings may give some guidelines for choosing appropriate system parameters for balancing the trade-off between survivability and cost.

Take the graph in Fig. 1 as an example. There are 11 machines represented by nodes and 5 distinct software products represented by different colors. We expect diverse software products developed by different people will not have the same vulnerability for the attacker to exploit [6], and an attack can propagate from one node to another by exploring one kind of vulnerabilities residing in a particular software product. By applying our algorithm on this graph, we find that even in the worst-case scenario a successful attack can only compromise four machines (in green) at most. This indicates that by optimally assigning software to connected machines in a non-adjacent manner, our algorithm can effectively reduce the epidemic effect of an attack.

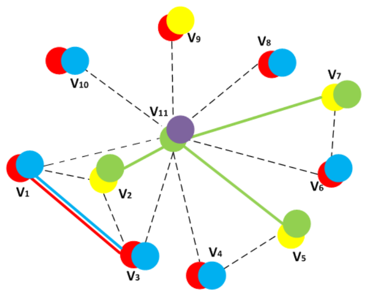

Fig. 1. Network topology utilizing a diverse software distribution. Solid lines connect nodes with the same color and indicate all possible worm-spreading paths; dashed lines connect nodes of different colors and indicate all non-spreading connections.

2 Related Work

Software Diversity The state-of-the-art approaches on software diversity are classified into three main categories: diversity at the system level and the network level, and N-version programming.

Software diversity at the system level has been researched to address space layout randomization, instruction set randomization and data randomization. Address space layout randomization (ASLR) [7] randomizes the memory address of the program region and has already been implemented in major operating systems [8]. The instruction set randomization (ISR) [9] obfuscates underlying system's instructions in order to defeat code-injection attacks. Data randomization [10] is another randomized-based approach, which applies masks on data in the memory, so that the attacker cannot determine memory regions that are associated with particular objects. Diversity at the network level is achieved by using different applications [4] [5], operating systems, and shuffling techniques within a networked system [11]. N-version programming was introduced in the 1970s which depends on functionally equivalent versions of the same program and monitors the behavior of the variants in order to detect divergence of the results [12].

Graph Coloring Graph coloring [13] is a famous problem in graph theory, which ensures there is no two adjacent nodes sharing the same color. Its solutions is a natural option in modelling a wide range of real-life problem for making globally optional decision [14] [4]. In most cases, however, the basic graph coloring problem fails to model some real-life problems where additional constraints have to be satisfied. Variants of the basic coloring problem have been introduced and intensively studied which allow one to take into account certain local constraints. List coloring [15], precoloring extension [16] and H-coloring [17] are examples of local constraitns. In some cases, these problem can be more difficult algorithmically than traditional vertex coloring [16] [18]. Another widely studied coloring problem is multicoloring problem [19] where more than one color have to be assigned to each vertex.

3 Problem Formulation

3.1 System Model

In this work, we use an undirected graph as the abstraction of a general finite networked system. Formal definition of the graph is given as follows.

Definition 1 (*Communication Graph*) A communication graph is an undirected graph $G = (V, E)$, where V is a finite set of nodes which represents hosts and devices comprising the networked system, and E as a set of edges represents the communication links (e.g., physically directly connected or can communicate through network e.g. using TCP/UDP) between two nodes.

Example networked systems include intranet, enterprise social networks, tactical mobile ad hoc networks, wireless sensor networks of different network topologies. We show an example of how to model a real-world network using communication graph of nodes and edges (Fig. 2). In this example, we have three servers reside in the DMZ, seven workstations and a database server located in the internal network. The links shown between workstations and servers depict direct communication between them. Corresponding communication graph of the network can then be generated based on their connectivity.

Considering the characteristics of the system model, edges of the communication graph can be classified into two types of links: *defective edge* and *immune edge*. *Defective edge* is a type of connection whose two endpoints have the same type of software installed (hence potentially share the same types of vulnerabilities. Note that our algorithm does not assume the knowledge of vulnerabilities. We discuss this more in Section 6). Otherwise, an edge is an *immune edge*. In this paper, based on the above definitions, we further define formally the concept of common vulnerability graph.

Definition 2 (*Common Vulnerability Graph*): A common vulnerability graph (CVG) S is a subgraph of a communication graph G, inside which all vertices share a common software and are connected through defective edges, whereas all its boundary edges are immune edges.

3.2 Graph Multi-Coloring Problem with Local Constraints

Using the model described earlier, we model the software assigning problem as a graph multi-coloring problem, where each machine is represented by a vertex and

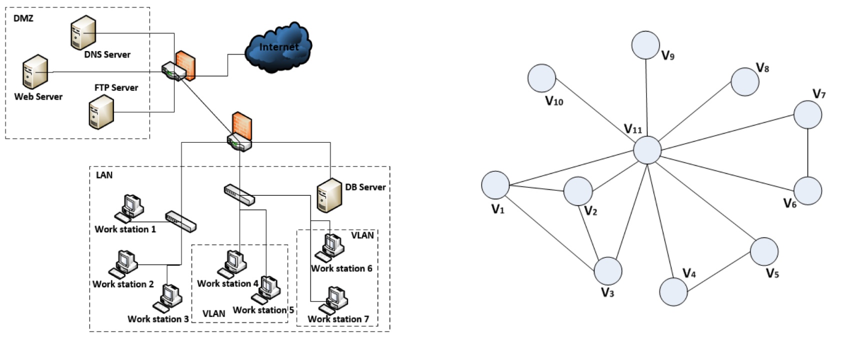

Fig. 2. Example of an enterprise network, and its corresponding communication graph.

each distinct software product is represented by a color. In the multi-coloring problem each node has a demand $w(v_i)$, which is the number of colors it requires. We further define $G = (V, E, W)$ to be a *weighted communication graph* with a weight vector W which denotes the number of software required for vertices V of the graph. A multi-coloring task of G is an assignment of colors to the vertices such that each vertex v_i is assigned with $w(v_i)$ colors in a way that two connected node cannot be assigned same colors.

However, practical software assignment contains several implicit constraints which basic multi-coloring problem fails to capture. For our work, we specifically consider two types of constraints that give rise to different system requirements, as defined below.

Definition 3 (*Constraint*): If a single host or a pair of hosts is restricted by some pre-defined software assigning rules, we say that the host(s) is/are constrained.

- *Host constraint*: certain hosts must be installed with some specified types of software to perform required functionality (e.g., to deploy a database server it is required to assign DB2);
- *Software constraint*: certain combination of software *must* or *must 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).

We note that these practical requirements could lead to the potential danger that nodes with the same software are neighbors, which violates the coloring rule that adjacent vertices have different colors. To better sketch our problem, we relax the requirement and re-define a less restrictive coloring rule that allows neighboring vertices to receive the same colors. The only thing we require is that the resulting CVG by that color has to be as small as possible. According to the definition, a defective edge indicates that the exploitation of one type of vulnerability on one host could lead to the compromise of its neighboring host following this "path" while immune edge naturally stops propagation of such automated attack. Thus we can use the size of CVG to indicate the infection range of an automated attack (e.g., worm attack). If one can effectively limit the size of the

CVG, system survivability can then be improved. Throughout the present paper, we denote the size of the largest size of the CVG as S, indicating the worst-case infection scenario. The ideal case is when every node in the network is isolated as a single CVG with size one (i.e. no defective edge). Hence, our ultimate research question becomes: *given a number of software products and certain constraints, how to load every node with software in a way that the size of the largest CVG is minimized?* One important assumption here is that vulnerabilities are unique for different software packages. Thus diverse software is vulnerable to different exploits and will not be compromised by the same attack. This assumption is confirmed by [6], in which it is found that more than 98.5% software have the substitutes and majority of them do not have the same vulnerabilities.

We are looking for a coloring assignment that satisfies above requirements and serves the goal of minimizing the largest CVG. Let C_h and C_s denote host and software constraints respectively. Next, we give a formal description of our software assignment problem.

Problem Given a weighted communication graph $G = (V, E, W)$ and a set of colors C, for each $v_i \in V$ assigns a subset $x(v_i) \subseteq C$, such that,

- v_i gets $w(v_i) \in W$ distinct colors, satisfying constraints C_h and C_s;
- for $\forall c \in C$, CVGs $S_i(c) \cap S_j(c) = \emptyset$, for any $i \neq j$;
- the size of the largest CVG is minimized.

The software assignment problem can be solved in either NP-time or P-time, depending on the given constraints. Our problem without considering constraints is a more general problem than ordinary vertex coloring, hence it is NP-hard in those cases where vertex coloring is NP-hard. And polynomial-time algorithms can be expected only in those cases where vertex coloring is polynomial-time solvable. We can expect that in certain situations, the special case of coloring is easier to solve than the general problem. Considering an extreme example when there are sufficient constraints that pre-define all the colors of vertices in G. In this situation, assignment solution can be obtained with $O(1)$. For some cases, however, not only our problem is NP-hard, but the case with constraint(s) is hard as well. In an example where there is only one constraint that pre-assigns a vertex v_1 with color red, the graph $G = (V, E, W)$ can then be updated to $G' = (V, E, W')$ by doing $w'(v_1) = w(v_1) - 1$. The problem then becomes multi-coloring on graph G', which is still NP-hard. It is worthwhile to study some restricted form of coloring with local constraints (e.g., give a bound on the number of constrained vertices), in the hope of finding a polynomial time solvable case. Since our problem can be NP-hard, optimal solutions may only be found when graphs are relatively small. Heuristics are needed to study large graphs. In the next section, we present a greedy algorithm to address the problem of software assignment with arbitrary constraints.

4 Software Assignment Algorithm

In this section, we present our algorithm for software assignment, which will produce an assignment of colors to vertices in a graph, subject to a set of constraints defined in the previous section.

4.1 Formulating Constraints

To incorporate host constraints to the algorithm, we introduce a collection of tuples $CSTR_h = \{(v_i, \varepsilon)\}$ where v_i is a vertex in the communication graph and ε is a (set of) color value(s) indicating that vertex v_i is constrained by color(s) in ε. For example, $C_h = \{(v_1, \{c_1, c_2\}),\ (v_3, \{c_5\})\}$, means vertices v_1 and v_3 are fixed with colors c_1 and c_2, and color c_5, respectively. Similarly, we denote software constraint using $CSTR_s = (s, \varepsilon)$, where s is the indicator of co-dependent colors and ε is a set of colors. When s equals 0, it states that the set of colors in set ε must not be assigned together to any vertex. In contrast, 1 means that colors in ε must be allocated simultaneously in order to perform certain functionality. For example, assume for an assignment characterized by $C_s = \{(0, \{c_1, c_2, c_4\}), (1, \{c_1, c_3\})\}$ on which c_1, c_2, c_3 represent three distinct software with equivalent functionality, one may prefer not to coloring the single vertex with any combination of c_1, c_2 and c_4, (i.e., assigning them simultaneously) while colors c_1 and c_3 must be assigned together to this vertex.

4.2 Algorithm Description

The algorithm consists two phases: a labeling phase and a coloring phase. In the labeling phase, each vertex is assigned a distinct number as its label and ordered based on that label. Following the labeling phase, the second phase of the algorithm colors vertices in the ordered list sequentially. It works by first scanning the vertices in a graph and load each vertex as many colors as possible while making sure not to create defective edges. It then visits all the vertices that have not been completely colored in order. Upon visiting a particular vertex, the algorithm assigns a color that leads to minimum CVG.

Phase 1–Labeling. We initialize a set UNCOLOR to contain all the ordered, uncolored vertices and we assume that the available colors in the palette C are suitably ordered as $C = \{c_1, c_2, \cdots, c_k\}$, where $k = |C|$. The main process then assigns every vertex a unique number from 1 to n as its label, where n is the total number of vertices in the network graph. Next, based on the numbers assigned, we order the vertices in UNCOLOR so that vertices with smaller labels are listed first. Several ordering heuristics are available to help accomplish the task of labeling: random ordering, increasing degree ordering, and decreasing degree ordering. Here we choose random ordering as the basis of our labelling. The randomness is intentionally designed so that one can run our algorithm once again to get a new color assignment solution by reordering the vertices. Certainly, an important property would be that the qualities of the assignments need to be stable. It means the largest CVGs resulted from different assignments are about the same size. The effect of the ordering will be evaluated in Section 5.

Phase 2–Coloring. The coloring task of phase 2 is done by two procedures: ColorVertexI and ColorVertexII. In the coloring phase, ColorVertexI first applies host constraint and assigns pre-determined colors to certain vertices while satisfying the software constraint. After that, it then successively colors the ordered vertices through the iterative processes. When a current color is determined, the algorithm scans the vertices in UNCOLOR sequentially checking to see if any of them can be colored according to the rules–1) no two adjacent vertices share the same color; 2) software constraints are satisfied by assigning the color. Meanwhile, ColorVertex I also checks the constraint sets to ensure that all the pre-defined constraints are satisfied by assigning the colors. Noted that followed by each coloring action, both the weight and UNCOLOR set need to be updated. When a vertex v_i is assigned the current color, the weight w_i corresponding to it will be decreased by 1 by doing $w_i = w_i - 1$. Those vertices with weight equals to 0 will be removed from UNCOLOR (i.e., UNCOLOR = UNCOLOR - v_i). If the vertex violates any constraints or its weight is larger than 0, then this vertex will remain in UNCOLOR for next iteration. The process of ColorVertexI stops when either the UNCOLOR set is empty or no feasible colors can be assigned to any vertices in UNCOLOR set.

Although ColorVertexI limits the number of defective edges to the minimum extent, there is a high probability that not all vertices get colored due the rigorous coloring constraints of the algorithm. To further color those remaining vertices, we propose ColorVertexII. As a supplement to ColorVertexI, ColorVertexII releases some of the hard requirements by allowing certain adjacent vertices to share the same color. However, with the overall goal of increasing the survivability of a networked system, such release should still follow certain principles. To be specific, instead of targeting at controlling the number of defective edges (keep it as 0), ColorVertexII shifts its focus to minimizing the size of the maximal CVG. For better understanding, we use an example to illustrate our point. Given two graphs in Fig. 3. The left graph has 3 defective edges ($\{v_1, v_7\}$, $\{v_3, v_7\}$, $\{v_6, v_7\}$) but none of them share a common vertex; the graph on the right contains 3 defective edges ($\{v_1, v_4\}$, $\{v_2, v_4\}$, $\{v_4, v_8\}$) but all share a common vertex v_4. Accordingly, the maximal infectable number in the left graph is 2 (assuming the attack starts from one node), whereas for the graph on the right, this number is 4. Hence, if an attack takes place on both graphs, the potential damage in the left graph is smaller than that in the other graph even though the left graph has a larger number of defective edges.

After ColorVertexI, if UNCOLOR set is empty, it returns a perfect coloring solution where no adjacent nodes share same colors. Otherwise, ColorVertexII is called after ColorVertexI finishes. It tries every color one by one in the UNCOLOR set and choose one with the least penalties. Since penalty occurs when defective edges appear, least penalties means that with the color assigned to the particular vertex, it forms up a smaller CVG compared to all other colors. This process repeats until UNCOLOR set is empty.

Algorithm 1 Color Assignment Algorithm

Input: (1) Graph $G = (V, E, W)$; (2) Available colors are ordered and represented by integers $1, 2, \cdots, k$; (3) Ordering ω of vertices in V; (4) Constraint sets C_h and C_s;
Output: A color assignment of k colors, 1 through k, to vertices of G represented by an array.

```
1: initialize array $X$
2: **for** $l \leftarrow 1$ to $n-1$ **do**
3:     pick unlabelled $v \in V$ at random
4:     $label(v) \leftarrow l$
5: **end for**
6: UNCOLOR $\leftarrow V$
7: UNCOLOR $\leftarrow$ ApplyConstraint($G, C_h$)
8: **for** $i$ from the smallest to largest integer (color) **do**
9:     **for** each vertex $e$ in UNCOLOR **do**
10:        ColorVertexI($G, C_s, i, e$)
11:    **end for**
12: **end for**
13: **if** (UNCOLOR == $\emptyset$) **then** exit
14:     **for** each vertex $e$ in UNCOLOR **do**
15:         ColorVertexII($G, C_s, e$)
16:     **end for**
17: **end if**
18:
19: **procedure** APPLYCONSTRAINT($G, C_h$)
20:     **for** for each vertex $j$ related by constraint $c_h(j) \in C_h$ **do**
21:         color($X[j]$) $\leftarrow c_h(j).\varepsilon$
22:         update $w_j$ and UNCOLOR
23:     **end for**
24:     **return**(UNCOLOR)
25: **end procedure**
26:
27: **procedure** COLORVERTEXI($G, C_s, i, e$)
28:     **for** each vertex $r$ that directly connects to $e$ **do**
29:         check $L(r)$ and $C_s$ to see if $i$ is a valid color for $e$
30:         If so, color($X[e]$) $\leftarrow i$
31:         update $w_e$ and UNCOLOR
32:     **end for**
33: **end procedure**
34:
35: **procedure** COLORVERTEXII($G, C_h, e$)
36:     size = $\infty$
37:     Select color $icolor$ that results in the smallest CVG
38:     color($X[e]$) $\leftarrow icolor$
39:     update $w_e$ and UNCOLOR
40: **end procedure**
```

4.3 Algorithm Complexity Analysis

Initially, n nodes in graph G are available for coloring. In ColorVertexI, as all available colors (k) have to be tentatively assigned to every node, so there are $n * k$ checks in this step. Suppose there are n remaining uncolored nodes in the worst case, each node with w_i colors needs to be assigned, in ColorVertexII we need $n * w_i * k$ rounds to pick the optimal color for each of them. To satisfy the algorithm constraints, after each color allocation, ColorVertexII also needs to check the size of the CVGs containing the current node in order to find the optimal color assignment with the minimal value. This makes the time complexity of ColorVertexII $O(n * k + n * w_{avg} * n_i * k)$, where w_{avg} is the average number of weights for all of the nodes in the network, and $O(n^2 * k)$ in the worst case (the expanded vertex set size for each color in ColorVertexII is as large as the entire graph when n_i becomes n).

It is easy to show that if colors are fully available at each vertex during the coloring phase of our algorithm, i.e., $|L(v_i)| = k$, and the maximum degree in the

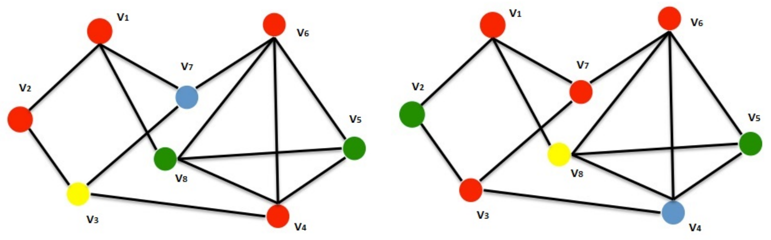

Fig. 3. Two random graphs with different software assignments

graph $\Delta = max_{0 \neq i \neq N} d(v_i)$, an optimal software assignment can eliminate the existence of defective edges if the number of available colors for the to-be-colored node $k \geq \Delta + 1$. This matches the well-known conclusion in graph coloring that, the chromatic number of a graph is at most $\Delta + 1$ [20]. It is also practically useful to derive a theoretical lower bound of $m = |C|$ on an arbitrary graph for a software assignment solution. However, it is difficult to analyze its lower bound when the system requirements and constraints vary across the network without restricting the constraints. For the case where there is no constraints, the lower bound of chromatic number had been discussed and proved [21]. Another extreme case is when all vertices of graph have been completely colored by pre-defined constrains. In this case, the chromatic number is exactly the number of colors of the graph.

5 Evaluations

In this section, we present simulation results for evaluating the performance of our method with respect to the above-mentioned requirements.

5.1 Simulation Setup

Recall that our model is built on top of undirected graph G as the abstraction of networked systems (Section 2.1). To fully investigate the performance of our algorithm in arbitrary systems, we use three representative topologies to characterize the behaviors of different systems. Specifically, we consider three types of graphs with different degree distributions: random graph, regular graph and power-law graph. A regular graph does not have high connectivity in many cases and thus long and circuitous routes are required to reach other nodes. Typical examples of regular graphs includes lattice and ring lattice graph, which can be used to characterize the behavior of each vertex that depends upon the behavior of its nearest neighbors. In a random graph, each pair of vertices are connected with the same probability and the degrees of each vertex are distributed according to a binomial distribution. In a power-law graph, the degree distribution satisfies a power law. It contains highly connected nodes (hubs) and is a good model for the highly connected systems. Scale-free networks follow power-law and many well-known networks such as WWW and social networks are believed to be scale-free [22]. We believe these three types of graphs provide a reasonable coverage range of realistic networked systems.

In this simulation, we generate graphs using *igraph* package in R with approximately the same average degree of 8. Given approximately the same degree settings across tested graphs, we observe if network connectivity affects the defense capability of our method. The default size for graphs in our experiment is 1000. Since our proposed mechanism only targets on networked systems with central authority (so that assigning software could be possible), the size of network chosen for experiment is reasonable and adequate to illustrate the results of our study. All simulation results here are presented as the average of 50 trials.

To evaluate our approach under various settings, we run the simulations with different combinations of the number of colors and weights, which is denoted as $\#color$ and $\#weight$, respectively. The total $\#color$ in the "color pool", represents the unique number of software choices available for hosts to choose from, and $\#weight$ represents the average number of distinct software finally installed on each host. We further define $r = \#weight/\#color$. In our simulation we set $\#color$ to 30 by default. We believe 30 is a reasonable number to illustrate the properties of our algorithm (Section 6 will discuss how to reduce the number of colors for a real system). In order to see how the number of average weight impacts the performance metrics, $\#weight$ is set to be an integer value ranging from 1 to 20. Intuitively, the larger r, the less heterogeneity of the system. We avoid choosing values larger than 20 because when it exceeds 20, the whole system becomes almost homogeneous.

5.2 Simulation Metrics

We adopt two metrics while evaluating the performance of our approach, including 1) the maximal number of nodes that can be possibly compromised; 2) the average size of isolated CVGs. For the first metric, we define S as the number of nodes contained in the largest CVG, denoting the number of machines compromised under the worst-case attacks. Nodes within the maximal CVGs are always attackers' first choice to penetrate into the network. In addition to that, we also consider the average size of CVGs, which indicates the overall robustness of the system. Although solely depending on the average size of CVGs may not directly indicate the survivability of the system, when taking it into considerations together with S, it can somehow be used to present separate infections caused by an attack. We use a symbol $\overline{s}$ to denote the average size of CVGs in the system.

5.3 Simulation Results

Impact of r. First, we use the power-law graph as an example to show how $r = \#weight/\#color$ impacts the heterogeneity of a system. Fig. 4 shows the variation of the largest CVG size S in responding to the changes of $\#weight$ and $\#color$ in a power-law graph.

We observe that given the same $\#color$, when $\#weight$ increases, the CVG size S becomes greater. In addition to the general distribution, we also notice that all three lines with different $\#color$ generate relatively flat trends at the beginning and are followed by a sharp increase when r reaches 0.4. Before this critical point, S remains relatively small (i.e., when $r < 0.4$, S is less than 50). However, after r exceeds 0.4, S begins to increase rapidly.

The simulation results indicate that the degree of heterogeneity our algorithm can create actually depends on the value of r. In general greater r tends to generate larger CVG, which in turn indicates a more homogeneous system. For instance, when $\#weight$ equals to $\#color$ ($r = 1$), there is only one component in the graph (the worst case). The results also suggest very limited decrease of the CVG size when r is less than 0.4. Hence, for systems already with a r value less than 0.4, there is little need to reduce the $\#weight$ (e.g., by decreasing the number of software packages to be installed in each host) to prevent automated attacks.

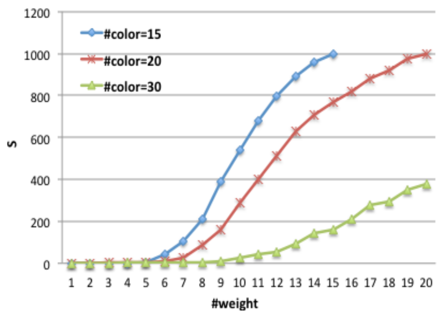

Fig. 4. S as a function of $\#weight$ and $\#color$.

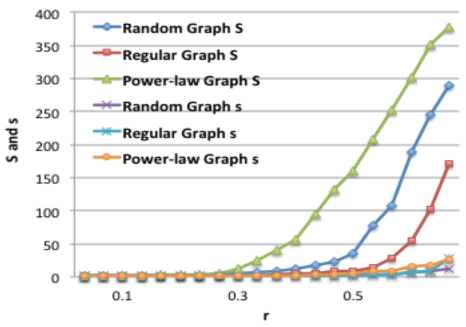

Fig. 5. Performance on three representative graphs.

Impact of Network Topology. Next we conduct simulation to see how different network topologies might affect the CVG size (S) and the average size of isolated CVGs ($\overline{s}$). As can be seen from Fig. 5, although the same "flat to sloping" trends are also observed in all three different topologies, we find that the CVG size of a power-law graph actually starts to increase at a relatively low r value (0.4) as compared to the cases of a random graph (when $r = 0.5$) and a regular graph (when $r = 0.6$). The difference in "turning point" is due to the distinct connectivity of each kind of topology. Given the existence of high connectivity nodes in a power-law network, large CVGs tend to be more easily formed than in the other two types of graphs. In contrast, as characterized by low connectivity and relatively high cliquishness [23], a regular graph only generates large CVGs with larger r.

Besides the measurement on the CVG size, we also monitor the average size of all CVGs $\overline{s}$. Unlike our prior observations, we find that the distribution $\overline{s}$ with r increases gently this time. Although there also appear to be some turning points in all three distributions, even if r exceeds the threshold, $\overline{s}$ remains relatively small values (when r reaches 0.6, $\overline{s}$ of three graphs is still smaller than 50) as compared to the CVG sizes. The explanation of this phenomenon is that, when a CVG is formed (after r exceeds the threshold), the sizes of CVGs become

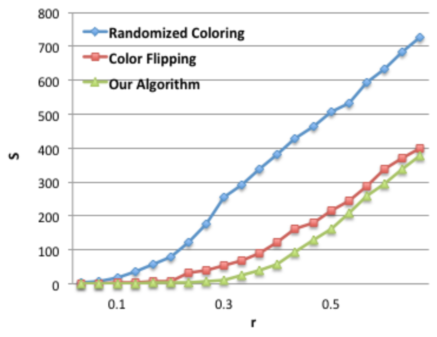

Fig. 6. Comparison with other algorithms

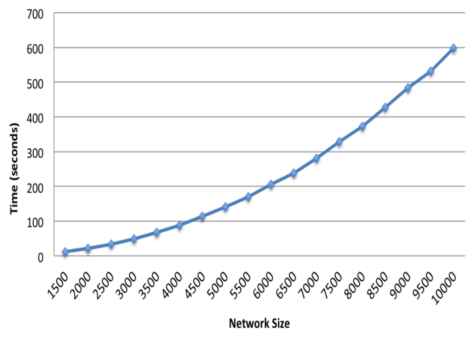

Fig. 7. Time overhead (in seconds)

polarized. That is, except the maximal ones which are very large, the other CVGs are relatively small (sizes between 1 to 50).

The above results are useful as they suggest that organizations with limited budget or software availability can still enhance their system heterogeneity by changing the underlying topology if possible.

Comparison with Other Algorithms. We next compare our algorithm with two related algorithms, randomized coloring and color flipping, as proposed in a previous study [5]. In randomized coloring, each node picks a tentative color uniformly at random from the color pool, whereas color flipping extends the random coloring by allowing each node performs a local search amongst its immediate neighbors to switch colors to decrease the number of locally defective edges. Besides, we also compare our algorithm with the optimal solution based on brute-force search. We compare these algorithms in terms of the CVG size (S) as a function of r. Results are plotted in Fig. 6. As we can see from the figure, our algorithm outperforms both randomized coloring and color flipping methods by creating smaller CVGs given the same r.

Scalability and Computational Overhead. Finally, we repeat the experiment on graphs of larger size and measure the computational overhead introduced by our algorithm. Even though Fig. 7 shows the average time required by this assignment algorithm grows exponentially with size of the network, it still suggests our algorithm can be applied to large systems with thousands of nodes with acceptable time overhead. As showed in Fig. 7, in a commodity PC, it takes about 10 minutes to assign colors to 10 thousand nodes. The simulation result confirms the practicability of our algorithm.

6 Discussion and Future Work

Source of Diverse Software. In our problem setting, software diversity could be caused by a number of factors. First, installing different types of software on adjacent machines is itself a type of software diversity. One might doubt that if enterprises are willing to invest amounts of resources (e.g., purchasing licenses, training employees) to realize the mechanism. However, there is always a

trade-off between security and costs. We believe the insights from the simulation results could help to ease the issue of finding the balance. Second, software of the same type might be implemented independently. For example, a number of web browsers are freely available, e.g., Firefox, IE, Chrome, so we may assume their vulnerabilities are different [6], and consider them as different colors. Third, with new compiler technique [24], each download of a piece of software could be a variant version of the client software which may not share the same vulnerability.
Vulnerabilities in Software Another thing needs to be clarified in the context is that we do not actually need full knowledge of vulnerabilities (e.g., types and the exact number) in the software to be assigned. Given a software assignment solution, no matter how many vulnerabilities residing in a software product, even if it has an unknown vulnerability, CVGs of the graph remain the same. For the same reason, we can further shrink the size of "color pool" by considering multiple software packages as one single software product that can be represented by using a single color. For instance, all the standard software packages (including network services) come with Windows 7 in a batch, so the entire package of the standard Windows 7 distribution will only considered as one color. The edge between two adjacent hosts is defective as long as they both install Windows 7. For each (third-party) application installed later on, it may be considered as one color if it provides some network services.
Severity Level of Vulnerability. In reality, vulnerabilities often have different levels of severity. The risks of some vulnerabilities are minor while some may be quite significant. For two CVGs of the same size, a more severe vulnerability (e.g., gain complete control of a system) could result in a far more serious damage compared to a mild vulnerability (e.g., allow an attacker to collect some types of user information). As such, we may use different values to represent different levels of damage that could be resulted from an attack, and our objective of optimization will then become to minimize the overall damage by the attack. In our future work we will devise a new algorithm by taking into account this dimension.

7 Conclusions

In this work, we proposed a method for effectively containing automated attacks via software diversity. By building up a heterogeneous networked system, our defense mechanism increases the complexity of the networked system by utilizing off-the-shelf diverse software. Given the practical problems of software assignment, we presented a software assignment algorithm based on graph multi-coloring with real world constraints and system prerequisites. We analyzed the effectiveness of our methodology through extensive simulation study.

References

1. Lala, J.H., Schneider, F.B.: It monoculture security risks and defenses. Security & Privacy, IEEE **7**(1) (2009) 12–13

2. Stamp, M.: Risks of monoculture. Communications of the ACM **47**(3) (2004) 120
3. Zhang, Y., Vin, H., Alvisi, L., Lee, W., Dao, S.K.: Heterogeneous networking: a new survivability paradigm. In: Proceedings of the 2001 workshop on New security paradigms, ACM (2001) 33–39
4. Yang, Y., Zhu, S., Cao, G.: 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 (2008) 149–158
5. O'Donnell, A.J., Sethu, H.: 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 (2004) 121–131
6. Han, J., Gao, D., Deng, R.: On the effectiveness of software diversity: A systematic study on real-world vulnerabilities. Detection of Intrusions and Malware, and Vulnerability Assessment (2009) 127–146
7. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Security and Privacy (SP), 2013 IEEE Symposium on, IEEE (2013) 574–588
8. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: Enhanced operating system security through efficient and fine-grained address space randomization. In: USENIX Security Symposium. (2012)
9. Davi, L.V., Dmitrienko, A., Nürnberger, S., Sadeghi, A.R.: Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and arm. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, ACM (2013) 299–310
10. Philippaerts, P., Younan, Y., Muylle, S., Piessens, F., Lachmund, S., Walter, T.: Code pointer masking: hardening applications against code injection attacks. In: Detection of Intrusions and Malware, and Vulnerability Assessment. Springer (2011) 194–213
11. Jafarian, J.H., Al-Shaer, E., Duan, Q.: Openflow random host mutation: transparent moving target defense using software defined networking. In: Proceedings of the first workshop on Hot topics in software defined networks, ACM (2012) 127–132
12. Salamat, B., Jackson, T., Gal, A., Franz, M.: Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In: Proceedings of the 4th ACM European conference on Computer systems, ACM (2009) 33–46
13. Jensen, T.R., Toft, B.: Graph coloring problems. Volume 39. John Wiley & Sons (2011)
14. Chang, R.Y., Tao, Z., Zhang, J., Kuo, C.C.: A graph approach to dynamic fractional frequency reuse (ffr) in multi-cell ofdma networks. In: Communications, 2009. ICC'09. IEEE International Conference on, IEEE (2009) 1–6
15. Voigt, M.: List colourings of planar graphs. Discrete Mathematics **120**(1) (1993) 215–219
16. Hujter, M., Tuza, Z.: Precoloring extension. ii. graph classes related to bipartite graphs. Acta Mathematica Universitatis Comenianae **62**(1) (1993) 1–11
17. Bulatov, A.A.: H-coloring dichotomy revisited. Theoretical Computer Science **349**(1) (2005) 31–39
18. Tuza, Z.: Graph colorings with local constraints-a survey. Discussiones Mathematicae Graph Theory **17**(2) (1997) 161–228
19. Borodin, A., Ivan, I., Ye, Y., Zimny, B.: On sum coloring and sum multi-coloring for restricted families of graphs. Theoretical Computer Science **418** (2012) 1–13

20. Welsh, D.J., Powell, M.B.: An upper bound for the chromatic number of a graph and its application to timetabling problems. The Computer Journal **10**(1) (1967) 85–86
21. Bollobás, B.: The chromatic number of random graphs. Combinatorica **8**(1) (1988) 49–55
22. : Scale-free networks (https://en.wikipedia.org/wiki/Scale-free_network)
23. Premo, L.: Local extinctions, connectedness, and cultural evolution in structured populations. Advances in Complex Systems **15**(01n02) (2012)
24. Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., Franz, M.: Compiler-generated software diversity. In: Moving Target Defense. Springer (2011) 77–98