Security Informatics
a SpringerOpen Journal

RESEARCH Open Access

Diversity and resistance in a model network with adaptive software

Neal Holtschulte* and Melanie Moses

Abstract

Attacks on computers are increasingly sophisticated, automated and damaging. We take inspiration from the diversity and adaptation of the immune system to design a new kind of computer security system utilizing automated repair techniques. We call the principles of effective immune system design Scalable RADAR: Robust Adaptive Decentralized Search and Automated Response. This paper explores how node diversity is maintained on a network that can generate software variants at individual nodes and make local decisions about sharing variants between nodes. We explore the effects of different network topologies on software diversity and resource trade-offs. We examine how the architecture of the lymphatic network balances trade-offs between local and global search for pathogens in order to improve our design. Experiments are performed on model networks of connected computers able to automatically generate repairs to their own software in response to an attack, bug, or vulnerability. We find that increased connectivity leads to increased overhead, but decreased time to repair, and that small world networks more efficiently distribute repairs. Diversity is diminished by increased connectivity, but has a more complex relationship with network structure, for example, a highly connected network may exhibit low overall diversity but maintain high diversity in a small number of low degree nodes in the periphery of the network.

Introduction

In the realm of cyber security the attacker currently has the advantage. Defenders face a wide variety of constantly adapting threats, but a great deal of software and many operating systems are identical. Due to this monoculture, an attack that works against one computer will work against many. Software monoculture also encourages attackers by increasing attack scalability at no cost to the attacker, e.g. Microsoft Windows is not necessarily more vulnerable than other operating systems, but its large market share makes it a preferred target.

Animal immune systems also face an onslaught of diverse and adaptable attackers, yet effectively defend against disease and infection. Immune systems do so by being adaptable, robust, scalable, and diverse. Diversity is a valuable asset to a defender. Diversity prevents any one attack from compromising a large portion of the defender's systems. The system of automated software repair that we describe makes it possible to synthesize diversity and deploy software variants that will not be vulnerable to the same attacks, increasing difficulty and cost for attackers.

We call the principles of effective immune system design Scalable RADAR: Robust Adaptive Decentralized Search and Automated Response [1]. We seek to adapt these principles to the realm of computer security and tilt the balance of power in favor of the defender. Our goal is to automatically identify security vulnerabilities and attacks in software and repair them in real time at the very large scales required by real computing systems.

In this paper we simulate the detection of malicious inputs, repair of underlying bugs, and distribution of repairs on a variety of network topologies. This is not a model of how malware spreads, but rather we model how computers on a network can distribute patches or repaired variants of software in a fashion that does not require top down control or manual intervention. In the current implementation, the faults that need repaired are generic and represent bugs, vulnerabilities, exploits, or any other undesirable behavior, but the faults do not spread as computer worms do.

We build our model to investigate several key questions: What is the relationship between network topology and

*Correspondence: neal.holts@cs.unm.edu
Department of Computer Science, University of New Mexico, Albuquerque, USA

the speed with which nodes acquire immunity to attack? How is network overhead, in terms of the amount of software shared, related to the time to resist attack? Do some networks balance the tradeoff between software sharing and time to resistance better than others? Does the diversity of software in a network increase or decrease the time for nodes to acquire resistance to new attacks? What is the optimal level of diversity and how can it be maintained without sacrificing local response times?

We model different network topologies including rings, small world networks, a community structure network, and a binary tree, and subject each network to the same series of simulated "attacks". We measure the time between the start of each attack and the incorporation of a repair (either generated locally or shared by a neighboring node). We also measure the amount of overhead in terms of the number of software variants shared between nodes, and we measure the diversity of software on each network.

We hypothesize that networks with greater connectivity and shorter mean path length will more rapidly distribute repairs, but will decrease diversity in the process. This may prove short-sighted if decreased diversity makes it more difficult to find repairs to later attacks.

Small world networks are found widely in nature [2,3], but we hypothesize that networks with more isolated components (such as a binary trees or rings) will likely promote greater diversity in the same way that speciation can occur when a subset of a population becomes isolated from the rest of the population and is subjected to different fitness criteria [4]. The binary tree and ring networks, however, will take longer to acquire resistance as the limited information flow will force nodes to spend more time generating repairs locally.

Background and literature review

Principles from immunology

When faced with a deadly infection, the immune system must rapidly find and neutralize a small number of pathogens hiding among trillions of healthy host cells or the host dies. In [1] we propose a set of design principles used by immune systems, ant colonies and other complex biological systems. We identify mechanisms that have evolved for Scalable Robust, Adaptive, Decentralized Search and Automated Response (Scalable RADAR). These properties are relevant to computer security, where distributed, autonomous, rapid, robust and adaptive control networks are required to defend against increasingly sophisticated attacks. The immune system has evolved lymphocytes (B and T cells) to adaptively recognize and neutralize pathogens. Other immune cells carry pathogens to lymph nodes where lymphocytes can find them. The architecture of the lymphatic network that connects lymph nodes to each other and to tissue facilitates the search for pathogens and production of antibodies that neutralize them.

Immune system inspired approaches have been particularly successful in computer security (reviewed in [5]) where immune inspired intrusion detection are distributed, scalable and sometimes robust to small failures, but there has been little success in scalable automated response. We identify design principles that lead to scalable RADAR in the immune system as a foundation for developing architectures for computer security systems that mimic the principles of scalable RADAR.

Immune systems are **robust**. Degeneracy (partial overlap in the functionality of multi-functional components) and proportional response to threats both contribute to robustness. Components are degenerate such that if one cell dies, there are multiple similar cells to perform its task with some degree of competency. Immune systems are **adaptive** because populations of individuals change in response to environmental signals. For example, activated B cells produce a large and variable population of daughter B cells. Those that bind to pathogens most effectively reproduce faster, so the population of cells improves its ability to neutralize the pathogen.

Search in the immune system is **decentralized**. No cell tells the other cells what their task is, or when or where they should do it. Cells sense chemical signals, environmental stimuli, and rates of interaction with other immune cells to determine how, when and where to search. While control of the search for pathogens is completely decentralized, communication between individuals is aggregated spatially in lymph nodes that concentrate interactions between immune cells and pathogens to improve the search process. Immune cells **respond** to attacks by integrating local signals from their environment to determine their behavior. Some local responses may be 'errors', but the response of the whole system is governed by collective agreement.

Immune systems **scale** up to trillions of cells. Because cells respond only to local signals, each can act in parallel without need for information signals to travel to every individual for a search to be effective or a response to be initiated. However, our analysis suggests that scalable response requires more communication between individuals as the system grows. We hypothesize that scalable communication patterns are promoted by the physical architecture of lymphatic networks. The immune system balances the need for local detection of pathogens with a systemic response to attack by distributing immune cells across a semi-modular hierarchical system of lymph nodes connected by the circulatory and lymphatic networks. When we compare across animals from mice to horses, the average size of a lymph node and the number of lymph nodes both increase with animal size, but the increase is sub-linear, so that a horse has more and larger lymph

nodes than a mouse, but neither the increase in lymph node size nor the increase in number are as great as the increase in body size.

In this paper we draw from our understanding of these scalable RADAR principles, and we focus on testing which computer network architectures balance the need to maintain diversity while rapidly responding to systemic threats.

Patch distribution

Traditionally, patch distribution has been centralized and hierarchical. System administrators are responsible for pushing out patches to a network, or individuals are prompted to begin the update process by notifications which direct them to a site where the patch can be downloaded. These distribution methods naturally lead to problems as computer systems grow more complex. According to [6], "manually applied patches are not effective in countering worms because they require human reactions and they are usually slow and do not scale well."

Systems of any significant size have automatic patching and updating processes, but these systems still download software from a central source. Additionally, diversity in the code base is seen as a problem to be overcome rather than a resource to be utilized. In [7], a system administrator laments "We have over 100 UNIX systems running more than half a dozen UNIX based OSes (more than a dozen when counting different OS versions). ... all configured slightly differently to suit their particular users' needs. ... the majority of our days were spent merely fighting fires."

Most patch distribution and management research focuses on timeliness, orderliness, and control. But all these systems rely on manually generated repairs and centralized distribution of patches overseen by a system administrator. The ability to automatically generate and evaluate repairs allows us to take a fresh perspective.

Automatically evolving software that resists attack

Genetic programming (GP) is a biologically-inspired method of automatically creating or modifying software. GP uses operations such as mutation and crossover to evolve a population of software based on a user-defined fitness function.

Forrest and Weimer use GP to evolve variants of programs that are resistant to security vulnerabilities [8-11]. Their design uses swap, copy, and delete operators on program instructions in order to repair bugs while retaining the original program's required functionality. Both the fitness function and 'required functionality' are defined using test suites. Test suites are a common tool of software engineers. Test suites consist of correct input/output pairs that a program is expected to satisfy and are designed to test software correctness.

Forrest and Weimer have recently investigated the benefits of "synthesizing diversity" by generating neutral software variants. A software variant is said to be neutral with respect to a suite of test cases if it passes all the same test cases as the original program. Neutral software variants have been shown to repair bugs experimentally seeded into test programs [12]. When program variants evolved by GP were evaluated on the tests that had been removed from the original test suite, an average of 19 out of 5000 variants passed one or more of the tests. In other words, the variants had repaired some of the seeded bugs by chance. Based on these results, they introduce the idea of mutational robustness, that lightweight random changes to program code is relatively unlikely to discernibly change program behavior.

Unspecified behavior may not be exercised by test cases so mutations that affect such behavior may still be considered neutral by the above definition of neutral. Such mutations will only affect "fringe" behavior that is commonly exploited by malware. Neutral mutations can then be used to synthesize diversity and proactively protect against unknown bugs and novel attacks. "Normal" users may not even know that the software they are using is different from the software used by the person in the neighboring cubicle because the mutations are neutral with respect to the standard program behavior. Rather than viewing neutral mutants as an overhead to be avoided or an indication of test suite inadequacy (e.g. [13,14]), we propose that they enhance the evolutionary process and are useful in their own right as a source of proactive diversity.

Given that GP can be used to automatically generate software diversity, how should software be distributed so as to maintain diversity? In this paper we investigate the effect on diversity of distributing variants on different network topologies.

We seek to replicate the scalability, robustness, and adaptiveness of the natural immune system by mimicking scalable RADAR principles. By analogy with the lymphatic network, we focus on the design of the network that computers use to share repairs. In our model, repairs are shared locally among neighboring nodes instead of being managed by a centralized, global controller. We investigate network topologies that are conducive to RADAR properties. Our goal is to determine which network topologies have the fastest response to newly discovered bugs or exploits with the lowest overhead. To this end, we run experiments on each network topology for a large and a small network. Each experiment is run many times, simulating consecutive, increasingly problematic attacks, and we measure overhead, time to resistance, and diversity.

Research design and methodology

We ran two sets of six experiments on seven different network topologies. Networks were initialized with 1024

nodes in one set of experiments and 64 nodes in the other. Each experiment was run 100 times with different random seeds.

Each experiment consists of nine phases corresponding to the nine increasingly severe attacks used (see Table 1). At the beginning of each phase, every node is simultaneously subjected to the same attack. Each node that is not resistant to the attack sends requests for software variants to all its neighbors. Vulnerable nodes also attempt to generate their own repairs.

Nodes continue to generate software variants until a resistant variant is found or is received from a neighbor. Additional requests for variants are sent at regular intervals (see Table 2 for the specific number of time steps for intervals and actions).

Nodes respond to requests for variant software by transmitting a copy of their own software to the requesting node. Nodes always respond to requests for software variants regardless of whether or not they themselves are vulnerable to the current attack. This is wasteful since vulnerable software will be useless to the receiver, but it is only obviously wasteful because every node is subjected to the same attack at the same time. In future work this contrived attack pattern will be modified and nodes will not know a priori whether or not their neighbors' software will be effective.

The effectiveness of received software variants is not checked immediately upon receipt. Effectiveness is checked after the node has completed its current activity, such as generating its own repair. When a node that is not yet resistant to the current attack identifies a resistant software variant, the node replaces its own software with the variant. Incorporating a neighbor's software decreases diversity in the network, but the individual node no longer needs to spend its own resources attempting to generate a repair. This illustrates a tradeoff between local resources and global diversity.

Table 1 Attacks

Attack	Vulnerable space
0.9	79.5%
0.95	85.6%
0.98	91.0%
0.99	93.6%
0.995	95.5%
0.999	98.0%
0.9999	99.4%
0.99999	99.8%
0.999999	99.9%

Attacks are values in the range zero to one. Any software with a resistance less than the attack is considered vulnerable to the attack. The percentage of the entire resistance space vulnerable to each attack is given below.

Table 2 Constants

Constant name	Value	Unit
Attack interval	50000	time steps
Software edge traversal time	100	time steps
Software request edge traversal time	100	time steps
Software incorporation time	1	time step
Repair attempt time	100	time steps
Random variants tested per repair attempt	10	software variants
Software request interval	300	time steps
Node count	1024	nodes

The model simulates parallelism by associating an update time with all objects and incrementing update times by a specified amount when different actions are performed. Table 2 shows the time step penalties and other constants used in the simulation. Any actions not listed, such as a node checking its own vulnerability, incurs no delay. Attack interval is the number of time steps before the next attack occurs. There was never a case of a node being vulnerable to the previous attack when the next attack occurred.

Figure 1 shows a small ring network with three software variants (represented by small envelopes) enroute to neighboring nodes.

Each node on the network stores a 20-bit binary number used to represent the software that is attacked and, in response, repaired and distributed by the nodes. The nodes are initialized with identical, low-quality binary numbers. We will refer to these binary numbers simply as software or software variants.

The initial software is low-quality in the sense that it is vulnerable to almost any attack. These "attacks" represent either external attacks (infiltration by a malicious or unauthorized user), or bugs or vulnerabilities. They represent any sort of flaw discovered in the software that can, in principle be fixed using the GP techniques of Forrest and Weimer [11]. Each attack is a number in the range zero to one, with zero being the least severe and one being the most severe. In each experiment, the network is subjected to increasingly severe attacks. As the attack gets closer to one, fewer software variants exist that are resistant. Table 1 shows the nine attack values used for all experiments and the corresponding percentage of all possible values of 20-bit binary numbers that are vulnerable to these attacks. Any software with a resistance less than the attack is considered vulnerable.

A software variant (20-bit binary number), s, is converted to a floating point value, v, in the range zero to one as follows:

$$v = \frac{s}{(2^{20}) - 1}$$

where $(2^{20}) - 1$ is the maximum value that can be represented in a 20-bit binary string.

Software is initialized to the binary string most closely representing the value 0.547, which has one of the lowest

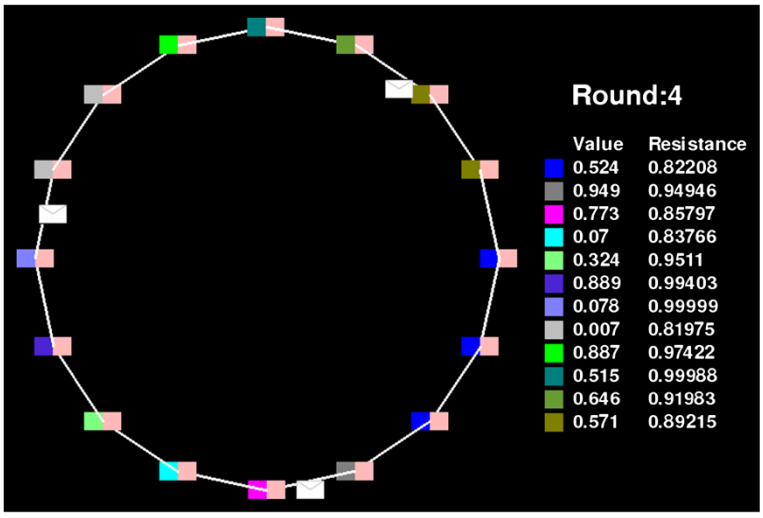

Figure 1 A screenshot of the visualization of software variant sharing on a small ring network with each node connected to its nearest neighbors. Software variants are represented by small envelopes. Three variants can be seen enroute. Nodes themselves are represented by bisected rectangles. The salmon color represents the attack and becomes increasingly red as the experiment moves through the attack progression. The other color in each rectangle is a unique color associated with each software variant. Values, colors, and resistances of these variants can be seen in the table on the right side of the image.

possible resistances in the middle of the range zero to one.

$$init = ((2^{20}) - 1) \cdot 0.547)$$

Resistance, r, for software with value, v, is calculated as follows:

$$r = \frac{sin(16 \cdot 2\pi \cdot v) + 1}{2}$$

This sine function has 16 optima in the range zero to one, a minimum y-value of zero, and a maximum y-value of one.

Figure 2 shows a snapshot of an experiment using a small number of nodes. The horizontal red line represents the value of the current attack. Green dots are representations of software on different nodes. The x coordinate of the nodes is the software value, the y coordinate is the resistance. The blue sine curve shows the resistance for values in the range. The five green dots beneath the red line show that five nodes are vulnerable to the current attack.

"Repairs" are automatically generated by naively testing random binary numbers for resistance against the current attack. If software is generated which resists the attack, then the node replaces its software with this new variant. Each attempted repair consists of testing ten randomly generated binary numbers.

The model simulates parallelism by associating an update time with all objects. At each time step, every object is checked to see if its update time is less than or equal to the global time. If so, the object is updated. For example, if node n has update time 100 and the current time is greater than or equal to 100, then n will check to see if it is vulnerable, check if any variants have been delivered from its neighbors, and, if enough time has passed since its last request for software variants, send another request. It takes one time step for a node to incorporate a software variant, but 100 time steps to make ten repair attempts. When a node makes these repair attempts, its time will be incremented by 100. Nodes respond to requests to share software without incurring any delay,

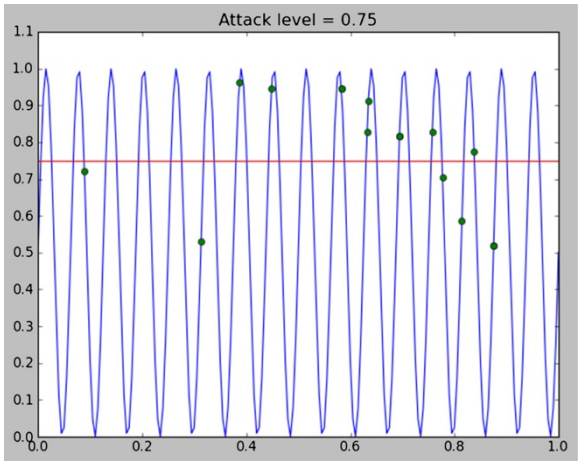

Figure 2 A snapshot of a dynamic graph showing nodes (green dots) and where their "software" falls on the resistance landscape (blue sine curve). As the attack (horizontal red line) rises, nodes must keep their software above the red line in order to resist the attack.

but both requests and software variants take time to traverse edges on the network. For more details, see Table 2. Any actions not listed, such as a node checking its own vulnerability incurs no delay.

Experiments were run on the following network topologies. All of these topologies were generated using the NetworkX module for python [15]:

- Ring k=1 (R1): A ring in which each node is connected to its two nearest neighbors (one on each side). The 1024 node R1 network has 1024 edges. The 64 node network has 64 edges. The 1024 node network was generated with the following call to the NetworkX module:

  ```
  networkx.connected_watts_strogatz_graph(1024, 2, 0.0)
  ```

  The connected Watts-Strogatz network forms a ring and holds this form when the rewiring probability is set to zero.
- Ring k=5 (R5): A ring in which each node is connected to its ten nearest neighbors (five on each side). The 1024 node R5 network has 5120 edges. The 64 node network has 320 edges.

  ```
  networkx.connected_watts_strogatz_graph(1024, 10, 0.0)
  ```

- Small World Ring (SWR) aka Newman Watts Strogatz network: A ring where each node, u, has three edges, one connected to each neighbor and one random long range connection. Addition of the long range edges turns the ring into a small world network in which every node is connected to every other node by a relatively short path. The 1024 node SWR network has 2048 edges. The 64 node network has 128 edges.

  ```
  networkx.newman_watts_strogatz_graph(1024, 2, 1.0)
  ```

- Small World Ring Rewired k=5 (RR): A ring in which nodes are connected to all their neighbors within a radius of five nodes just like the R5 network. However, RR is then 'rewired' such that edges are randomly chosen to be removed and replaced with edges that connects two nodes chosen uniformly at random. This mirrors the approach taken by [2] to investigate small world networks. Rewiring takes place as follows: With 5% probability an arbitrary endpoint of each edge is replaced with a node selected uniformly at random. Rewiring probabilities in the range 1% to 10% produce networks with the lowest mean shortest path with a minimum of long range connections [2].

  ```
  networkx.connected_watts_strogatz_graph(1024, 10, 0.05)
  ```

  Mean shortest path is calculated by taking the sum of the lengths of the shortest paths between all pairs of nodes in a network and dividing by the number of pairs. It is a common measure of small world networks, which are characterized by the small number of edges in the path between any two nodes. Small world networks have been popularized by the "Six Degrees of Kevin Bacon" game in which players try to connect an actor to Kevin Bacon through six or fewer co-star connections.
- Small World Preferential Attachment (PA): A random graph incrementally built up by preferential attachment. New nodes connect to existing nodes probabilistically, with greater weight given to existing nodes that already have many connections. The 1024 node PA network has 1023 edges. It does not have 1024 edges because the network is initialized with two nodes with one edge between them then 1022 nodes are added and one edge is added to connect each of the 1022 nodes. The 64 node network has 63 edges.

  ```
  networkx.barabasi_albert_graph(1024, 1023)
  ```

- Binary Tree (Bin): A hierarchical network, in the form of a complete binary tree. The 1023 node Bin network has 1022 edges. The 63 node network has 62 edges.

  ```
  branching_factor = 2
  height = int(math.log(1024, 2))-1
  networkx.balanced_tree(branching_factor, height)
  ```

- Caveman Graph (Cave): A modular network generated by making n cliques of size k. Then one node in each clique is rewired to connect to an adjacent clique. We generated caveman graphs with 16 cliques. Code for the caveman graph cannot be found in the current version of NetworkX, but can be accessed here [16].

  ```
  cliques = 16
  clique_size = 1024 / cliques
  networkx.connected_caveman_graph(cliques, clique_size)
  ```

For each experiment we measure the following:

- Prior Immunity: The chance that a node's current software is already resistant to a new attack. We count the times a node is immediately resistant to a new attack divided by the total number of new attacks against all nodes.

- Effective Shared: The chance that a software variant that node v receives from its neighbor resists the attack against v. We count the number of effective variants received divided by the total number of variants received. Only variants received while v is vulnerable are counted. Variants received after acquiring resistance are not counted towards the numerator or denominator. By ignoring late variants we will elevate the percentage of effective shared variants, but for this metric we are only interested in variants shared during the vulnerable phase.
- Total Software Sharing: a count of instances of software sharing between nodes over the course of the entire experiment. For this metric, unlike 'effective shared', we include software shared after the destination node has already achieved resistance. This metric measures network overhead.
- Average Time to Resistance: The average number of time steps between the start of an attack on a node and the node achieving resistance. This time could be zero if a node has prior immunity. Since nodes continually attempt to automatically generate repairs, average time to resistance is also a measure of the CPU overhead, the amount of CPU cycles a node spends generating variants.
- Diversity: We measure diversity using the Shannon Index

$$H' = -\sum_{i=1}^{S} p_i ln(p_i)$$

where S is the total number of distinct software variants and p_i is the probability that a node has variant i. That is, p_i equals the number of instances of variant i divided by the total number of nodes.

Results and discussion

All figures and data reported below are for 1024 node networks. The 64 node networks exhibited comparable results.

Prior Immunity: The chance that a node will be resistant to a novel attack is approximately 48% and was essentially constant across all network topologies. This value was the same whether there were 1024 or 64 nodes on the network. This is not surprising since the quality of a software variant is not evaluated along a continuum. It either is vulnerable to the current attack or it isn't. Even nodes on networks with greater diversity had the same chance of resisting novel attacks.

Total Software Sharing: The number of shared software variants is greater in networks with more edges. Figure 3 shows the total number of software variants shared between nodes. R5, the five-neighborhood ring, and RR, the rewired ring, have five times as many edges as the other networks (see Table 3). Consequently, they share between four and five times as many variants over the course of an experiment. The caveman network is not shown because it dwarfs the other columns. The caveman network has far more edges than any other network and consequently shares around 250,000 software variants.

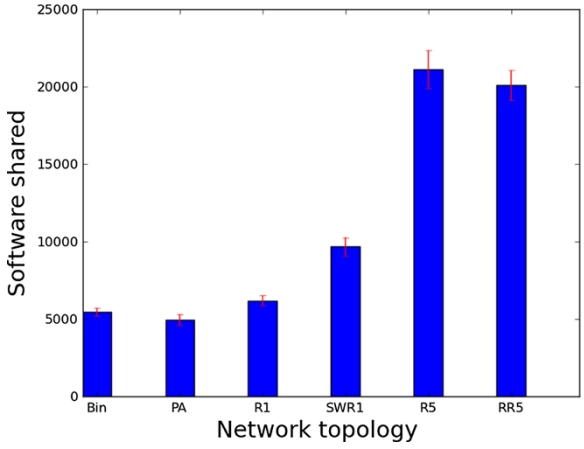

Figure 3 The average number of software variants shared during 100 iterations of each experiment for each network topology. This is a measure of the network overhead, which we wish to minimize. Software is shared when a neighboring node requests a variant in response to an attack. Network topologies are ordered from left to right by increasing number of edges. The data shown is for 1024 node networks. The caveman network is not shown because it dwarfs the other columns. The caveman network shares about 250,000 software variants.

Figure 3 also indicates that over the course of the nine attacks, on average eight to ten software variants are shared across each edge on each network, hence the network with 1024 edges shares about 10,000 variants and the RSW network with 5120 edges shares about 45,000.

Average Time to Resistance: The software sharing overhead should be viewed in the context of the time it takes nodes on these networks to acquire resistance to new attacks. Figure 4 shows that the network with the largest amount of software sharing is also the quickest to resist new attacks with Cave taking 6 steps on average. However, both PA and SWR buck the trend with low network overhead and modest average time steps to resistance.

Effective Shared: Figure 5 shows that the chance of receiving an effective variant depends on the network topology. More effective software variants are delivered on the random preferential attachment network than on any other, followed by the small world ring. One possible explanation for this effect for the random preferential attachment network is that few nodes in such networks contain the majority of the edges. Such nodes would be expected to quickly receive resistant software which they could then rapidly distribute to their neighbors.

Table 3 Network topologies

Topology	Edges	Nodes	Considered "small world"	Mean shortest path
Ring k=1	1024, 64	1024, 64	No	256.25, 16.25
Ring k=5	5120, 320	1024, 64	No	51.65, 3.66
Small World Ring	2048, 128	1024, 64	Yes	5.48, 3.11 *
Rewired Ring	5120, 320	1024, 64	Yes	5.27, 2.57 *
Preferential Attachment	1023, 63	1024, 64	Yes	2.00, 1.97 *
Binary Tree	1022, 62	1023, 63	No	14.07, 6.59
Caveman	32256, 96	1024, 64	No	9.00, 8.87 *

Two sets of experiments were run on six network topologies.
*denotes average values.

Diversity: Figure 6 shows the Shannon Index for each topology after each attack. The index after initialization is zero (not shown) because every node has the same software. The distance-one neighbor ring (R1) maintains the most diversity, which is easily explained by the relative difficulty with which any software variant would spread across this network. After R1, the binary tree (Bin) maintains the next highest diversity, suggesting that high mean shortest path corresponds to high diversity, which makes sense since longer path lengths limit the spread of software variants.

The 64 node networks exhibited comparable results to those reported above, not withstanding the dramatic difference in degree of the 64 node caveman network relative to the other networks. The chance of prior immunity, average time to resistance, and percentage of effective shared software was not significantly different between 64 and 1024 node networks with the same topologies.

Future work will look at more intelligent software sharing paradigms. For example, nodes may simply ignore a percentage of requests or we may use an economic model such as the one for reducing the bandwidth overhead on P2P networks introduced by [17]. Though far from the only pertinent feature of peer-2-peer networks, the preferential attachment network does have the same degree distribution (power law). Likewise, the caveman graph has a modular structure. Community structure is characteristic of many networks found in the real world such as P2P networks [18]. In future work, we will examine more realistic topologies with both community structure and power law degree distributions.

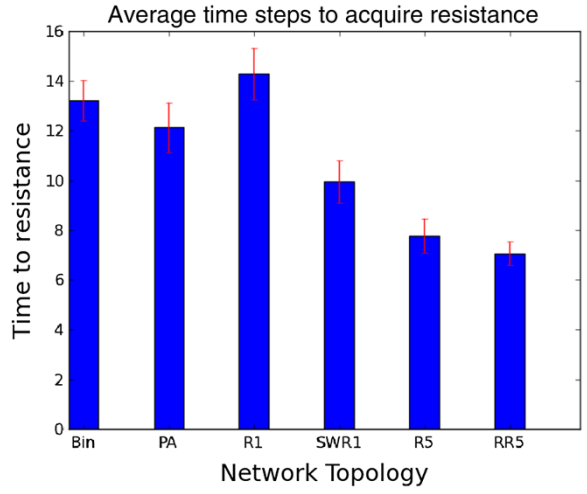

Figure 4 Average number of time steps between novel attacks and individual nodes' resistance to these attacks. These numbers are averaged over 100 iterations of each experiment for each network topology. Since one set of attempted repairs takes 100 time steps, this shows that on average no more than two sets of attempted repairs are made before a repair is generated locally or a resistant variant is received from a neighboring node. Network topologies are ordered from left to right by increasing number of edges. The data shown is for 1024 node networks.

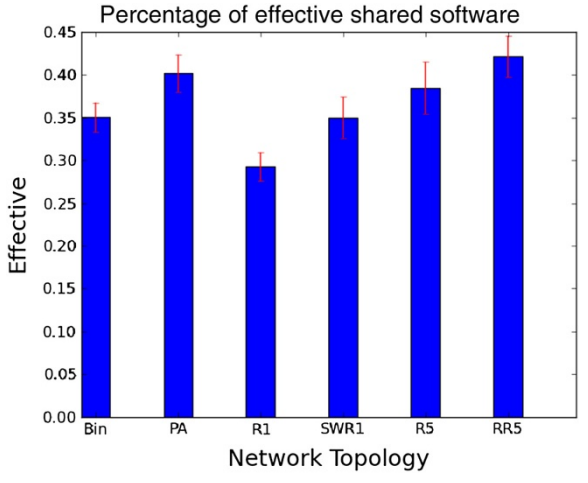

Figure 5 Percentage of shared software variants that effectively resist the receiving node's current attack. Only variants received while the node was vulnerable were counted. Variants received too late were not counted. Network topologies are ordered from left to right by increasing number of edges. The data shown is for 1024 node networks.

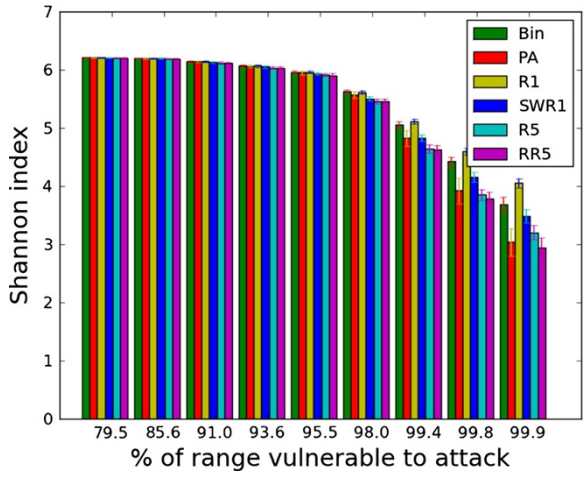

Figure 6 Average Shannon Index (a measure of diversity) over 100 iterations of each experiment for each network topology. Topologies are ordered from left to right by increasing number of edges. R1 stands for the ring network in which each node is connected to its nearest neighbors. R5 stands for the ring network in which each node is connected to its five nearest neighbors. SWR stands for the small world ring with one random long-range connection. RR stands for the rewired ring. PA stands for random preferential attachment. Bin stands for binary tree. The data shown is for 1024 node networks.

We will also add realism by replacing simulated bug fixes with actual repairs of bugs in open source C code. By performing real repairs we can increase the realism in a variety of ways, for example, the timing of variant sharing can be based on the number of clock cycles elapsed while generating the repair.

Conclusions

In this paper we simulate the detection of malicious inputs, repair of underlying bugs, and distribution of repairs on a variety of network topologies. We measure the speed with which nodes acquired immunity to attack, the overhead in terms of amount of software shared, and the diversity of variants that emerge on the network without any central control or distribution. We modeled the following network topologies: rings, a Watts-Strogatz small world network, a network formed by preferential attachment, a binary tree, and a so-called caveman network.

The total number of software variants shared over the course of an experiment directly corresponds to the number of edges in each network with eight to ten variants shared over each edge. The networks with more edges (R5, RR, Cave) show less sharing per edge. This may be due to the fact that more software sharing speeds up the acquisition of resistant variants, which stops nodes from requesting additional variants and therefore reduces software sharing. In short there is a negative feedback mechanism involved in software sharing in response to an attack.

The number of shared software variants for a given network topology is inversely proportional to the average time steps between a novel attack on a node and the node's resistance to the attack. In fact, Figure 3 is nearly the mirror image of Figure 5. It's interesting to note that the random preferential attachment network's average time to resistance is slightly lower than that of R1 and Bin, despite the fact that these graphs have the same number of edges and similar amounts of software shared. This can perhaps be explained by Figure 5, which shows that the chance that a shared software variant resists the current attack is highest on the preferential attachment network.

The fact that there is a difference in shared software quality on the different networks at all is surprising since there was no difference in prior immunity. The structure of these networks makes it possible for them to distribute resistant software more effectively without ever comparing software variants directly.

In the rings, R1 and R5, all nodes are created equally, with the same degree and all positions in the networks being equivalent. In all the other networks, except for the binary tree, there is a wide range in node degree. Nodes with higher degree will receive a larger number of software variants in response to any request. These high degree nodes may then distribute this software to all of their neighbors, resulting in the increased effectiveness seen. Recall that 'effectiveness' refers to the proportion of shared variants that resist the current attack level.

Small world networks generated by preferential attachment are characterized by robustness to the removal of random nodes, but quickly become disconnected if high degree nodes are removed. In other words, PA networks possess nodes with high "betweenness" through which the majority of the information traffic must pass. Betweenness is a measure of the probability of a node lying on a randomly chosen shortest path between nodes. In future work, we will investigate whether these high-betweenness nodes are responsible for the increased percentage of effective software sharing. Future experiments will address this question by examining the relationship between time to resistance and degree, distance to a highly connected node, and distance to the root in the binary tree. We will also look at the number of effective variants traveling up the binary tree (towards the root) versus the number traveling down towards the leaves.

Diversity as measured by the Shannon Index is largely a function of the mean shortest path of a network, but there is more to it than that. The random preferential attachment networks maintain high diversity as measured by a raw count of distinct software variants (see Figure 7), but have relatively low Shannon diversity, which also takes into account the number of each variant present on the

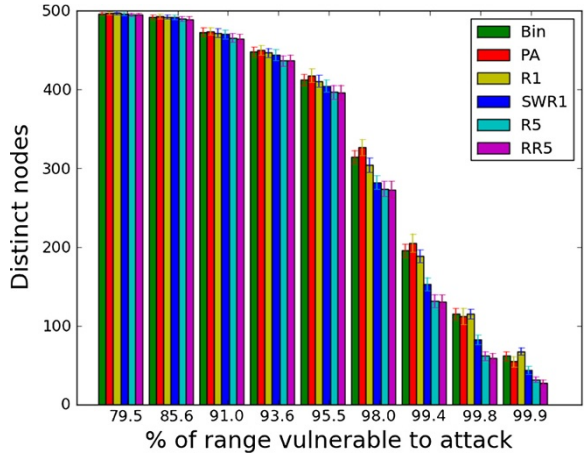

Figure 7 Diversity measured as a count of unique software variants on the network. These counts were averaged over 100 iterations of each experiment for each network topology. The preferential attachment (PA) network has a higher count of distinct nodes relative to the other networks than would be expected based on Figure 6.

network. This suggests that while PA networks are dominated by relatively few variants, corners of the networks maintain many unique variants. This is encouraging since we believe that diversity can be leveraged for increased robustness. It may be possible to implement different software sharing policies to encourage greater diversity on this and other networks. We leave this for future work.

In the immune system, the structure of the lymphatic network through which immune cells communicate is critically important for scalable RADAR. The precise topology of the lymphatic network is not known, but it is known that the network enables both local and global communication between lymph nodes, and as the number of lymph nodes increases, each lymph node communicates more [1]. This is thought to balance the needs for fast local search and systemic response. In this paper we have analyzed network topologies to determine which topologies promote diversity and rapid patch distribution with minimal overhead. In summary, overhead is proportional to the number of edges, but inversely proportional to time to resistance.

We find encouraging evidence that some network topologies allow effective patches to be widely deployed while still maintaining patch diversity. For example, effective patches are easily shared among nodes in small world networks. Furthermore, networks formed by random preferential attachment have low software diversity, but a high number of distinct software variants compared to the other topologies. In other words, there is low diversity in the majority of nodes, but high diversity in a minority of nodes. We believe this to be due to the ability of preferential attachment networks to distribute variants rapidly through their high degree nodes, reducing Shannon diversity of the overall network, while diverse nodes maintain a foothold in the low degree nodes at the periphery of the network. This is a desirable feature because if the majority of nodes are vulnerable to a new attack, it is likely that these nodes can be "recolonized" by a variant waiting in the wings. These results hold across multiple orders of magnitude difference in network size from 64 to 1024 nodes. Identifying the role of network topology in maintaining diversity and rapid response is a step toward developing more robust distributed computer security systems that mimic the adaptive qualities of the natural immune system.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
NH and MM contributed to the writing and analysis that went in to this paper. Both authors read and approved the final manuscript.

Acknowledgements
We thank Soumya Banerjee for helpful discussions and insights. This work is supported by grants from the National Institute of Health (NIH RR018754), DARPA (P-1070-113237) and National Science Foundation (NSF EF 1038682).

Received: 5 January 2012 Accepted: 7 September 2012
Published: 6 November 2012

References

1. Moses M, Banerjee S: **Biologically Inspired Design Principles for Scalable, Robust, Adaptive, Decentralized Search and Automated Response (RADAR).** In *IEEE Symposium Series in Computational Intelligence 2011 (SSCI 2011)*. Paris, France:2011.
2. Watts DJ, Strogatz SH: **Collective dynamics of 'small-world' networks.** *Nature* 1998, **393:**440–442.
3. Kleinberg JM: **Navigation in a small world.** *Nature* 2000, **406:**845.
4. Rundle HD, Schluter D: **Natural Selection and Ecological Speciation in Sticklebacks.** *Adaptive Speciation* 2004, **19**(3):192–209. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.3225&rep=rep1&type=pdf.
5. Forrest S, Beauchemin C: **Computer immunology.** *Immunological Rev* 2007, **216:**176–197. http://dx.doi.org/10.1111/j.1600-065X.2007.00499.x.
6. Castaneda F, Sezer EC, Xu J: **WORM vs. WORM: preliminary study of an active counter-attack mechanism.** In *Proceedings of the 2004 ACM workshop on Rapid malcode,* WORM '04. (ACM, New York, NY, USA; 2004:83–93. http://doi.acm.org/10.1145/1029618.1029631.
7. Ressman D, Valdes J: **Use of CFengine for automated, multi-platform software and patch distribution.** In *In USENIX 14th System Administration Conf. (LISA)*. USENIX Association, Los Angeles; 2000. pp. 207–218.
8. Fast E, Le Goues C, Forrest S, Weimer W: **Designing better fitness functions for automated program repair.** In *Proceedings of the 12th annual conference on Genetic and evolutionary computation* GECCO '10. ACM, New York, NY, USA; 2010. pp. 965–972. http://doi.acm.org/10.1145/1830483.1830654.
9. Forrest S, Nguyen T, Weimer W, Le Goues C: **A genetic programming approach to automated software repair.** In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation,* GECCO '09. ACM, New York, NY, USA; 2009. pp. 947–954. http://doi.acm.org/10.1145/1569901.1570031.
10. Nguyen T, Weimer W, Le Goues C, Forrest S: **Using Execution Paths to Evolve Software Patches.** In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops,*ICSTW '09. IEEE Computer Society, Washington, DC, USA; 2009. pp. 152–153. http://dx.doi.org/10.1109/ICSTW.2009.35.
11. Weimer W, Nguyen T, Le Goues C, Forrest S: **Automatically finding patches using genetic programming.** In *Proceedings of the 31st*

International Conference on Software Engineering, ICSE '09. IEEE Computer Society, Washington, DC, USA; 2009. pp. 364–374. http://dx.doi.org/10.1109/ICSE.2009.5070536.

12. Schulte E, Fry ZP, Fast E, Forrest S, Weimer W: **Software Mutational Robustness Bridging The Gap Between Mutation Testing and Evolutionary Biology.** 2012. http://arxiv.org/abs/1204.4224.
13. Offutt AJ, Untch RH: Mutation 2000: Uniting the Orthogonal. *Mutation Testing for the New Century*. Kluwer Academic Publishers, Norwell, MA USA; 2001. http://books.google.com/books?hl=en&lr=&id=LFvgCktM0sYC&oi=fnd&pg=PA34&dq=mutation+2000:+uniting+the+orthogonal&ots=pzA3SMn_EF&sig=1fo1swiEoFCvi5otIEk5zkRkalw.
14. Siami Namin A, Andrews JH, Murdoch DJ: **Sufficient mutation operators for measuring test effectiveness.** In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08: (ACM Press; 2008. pp. 351–360. http://doi.acm.org/10.1145/1368088.1368136.
15. **NetworkX: High productivity software for complex networks,** http://networkx.lanl.gov/. Accessed 20 April 2012.
16. **NetworkX, Network generators: Community,** https://bitbucket.org/bedwards/networkx-community/raw/370bd69fc02f/networkx/generators/community.py. Accessed 20 April 2012.
17. Roussopoulos M, Baker M: **CUP: Controlled Update Propagation in Peer-to-Peer Networks.** USENIX 2003 Annual Technical Conference 2002. p. 15 http://arxiv.org/abs/cs/0202008.
18. Porter MA, Onnela JP, Mucha PJ: **Communities in Networks. World Wide Web Internet And Web.** *Inf. Syst.* 2009, **56**(9):1082–1097. http://arxiv.org/abs/0902.3788.

doi:10.1186/2190-8532-1-19
Cite this article as: Holtschulte and Moses: **Diversity and resistance in a model network with adaptive software.** *Security Informatics* 2012 **1**:19.