Improving Sensor Network Immunity under Worm Attacks: a Software Diversity Approach*

[1]Yi Yang, [1,2]Sencun Zhu, and [1]Guohong Cao
[1]Department of Computer Science and Engineering
[2]College of Information Sciences and Technology
The Pennsylvania State University, University Park, PA 16802
{yy5,szhu,gcao}@cse.psu.edu

ABSTRACT

Because of cost and resource constraints, sensor nodes do not have a complicated hardware architecture or operating system to protect program safety. Hence, the notorious buffer-overflow vulnerability that has caused numerous Internet worm attacks could also be exploited to attack sensor networks. We call the malicious code that exploits a buffer-overflow vulnerability in a sensor program *sensor worm*. Clearly, sensor worm will be a serious threat, if not the most dangerous one, when an attacker could simply send a single packet to compromise the entire sensor network. Despite its importance, so far little work has been focused on sensor worms.

In this work, we first illustrate the feasibility of launching sensor worms through real experiments on Mica2 motes. Inspired by the *survivability through heterogeneity* philosophy, we then explore the technique of software diversity to combat sensor worms. Given a limited number of software versions, we design an efficient algorithm to assign the appropriate version of software to each sensor, so that sensor worms are restrained from propagation. We also examine the impact of sensor node deployment errors on worm propagation, which directs the selection of our system parameters based on percolation theory. Finally, extensive analytical and simulation results confirm the effectiveness of our scheme in containing sensor worms.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*; D.4.6 [**Software**]: Operating Systems—*Security and Protection*; K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems—*Security and Protection*

General Terms

Security, Algorithms, Experimentation, Performance

Keywords

Sensor Worm, Software Diversity, Graph Coloring, Percolation Theory, Sensor Network Security

1. INTRODUCTION

For sensor nodes operating in an unattended, hostile environment, they often face various security attacks. Due to their low manufacturing cost (e.g., less than one dollar for envisioned smartdust microsensors), it is unlikely that we will use expensive tamper-resistant hardware for these sensors. Therefore, malicious attacks are inevitable and they may be launched to obtain important information from sensor networks, to interfere with their normal operations, or even to destroy them. The question is how much we can do to increase the survivability of a sensor network that is under attacks.

In the literature, many security mechanisms have been proposed to defend against various kinds of attacks in sensor networks, for example, dodging communication channel jamming, thwarting MAC layer attacks, countering attacks against routing protocols, providing attack-resilient data aggregation [34], and node localization. However, we notice that a potentially more severe attack has not yet been studied. We name this attack *sensor worm attack*. More specifically, we define sensor worms as crafted messages that exploit software vulnerability of sensor nodes in a sensor network, causing sensor nodes to crash or taking control of sensor nodes. Clearly, sensor worm attack could be the most dangerous one if an attacker simply sends a single message to compromise the entire sensor network, defeating the mission of the sensor network. The only related work we know of is one on modeling sensor worm propagation [13]; it, however, neither discussed the feasibility of launching sensor worms nor proposed any solution to address sensor worms.

Naturally, the first question that will arise is: *is it possible for worm attacks to occur in sensor networks*? On one hand, compared with regular computer systems, it is even easier for sensors to be compromised by worm attacks. This is because sensors do not have complicated hardware architecture or operating system to protect program safety due to cost and resource constraints. Moreover, sensors in the same

*This work was supported in part by the National Science Foundation (CAREER-0643906, CNS-0524156, CNS-0519460, and CNS-0627382) and Army Research Office (W911NF-05-1-0270 and W911NF-07-1-0318).

MobiHoc'08, May 26–30, 2008, Hong Kong SAR, China.

network are homogeneous in both hardware and software. If one sensor is compromised because of a program vulnerability, all the other sensors are vulnerable to the same compromise attack. On the other hand, worm attack on sensors is not exactly the same as that on regular computers. The Harvard architecture of many sensor microcontrollers has separate memories for program and data. This prevents a piece of malicious code directly injected into the stack of the data space from being executed in such sensors. While this is generally true, as a trial study we conducted experiments on Mica2 motes and found that a buffer overflow vulnerability, the common trigger of worm attacks, could result in the transfer of program flow to a transmission component in the code space. Then, an exploited sensor may relay the attack packet it received before becoming irresponsive. Consequently, this leads to the propagation of the worm packet over the entire network and the failure/corruption of all the sensors.

The next question to be answered is: *what can we do to defend against such sensor worm attacks*? Although a huge body of literature exists on addressing the buffer overflow problem for protecting both clients and servers in the Internet, it is not immediately clear whether and how these solutions may be adapted to the sensor system, which features wireless communication, high connectivity, different hardware architecture, OS (e.g., TinyOS in Mica motes) and programming language (e.g., nesC for TinyOS). While this remains an interesting open research question, in this work we are more interested in improving the *survivability* of entire sensor networks under worm attacks.

In spirit of the *survivability through heterogeneity* philosophy, we explore the technique of *software diversity* to combat sensor worms. While the general idea of software diversity is not new and it has been applied to wired networks [17, 27, 9], its application to sensor networks faces unique challenges due to high node density and sensor deployment error. High node density implies that it is unrealistic to assign a different version to each node; deployment errors could lead to the potential danger that sensor nodes with the same version of code become neighbors (connected) after they are deployed into the field.

To address the above challenges, we first adopt a location-based version assignment strategy. Given a limited number of software versions of the same functionality, we load every sensor with a proper version of the software through an efficient graph color assignment algorithm, such that a sensor worm may be isolated in a small "island". Then we analyze based on percolation theory the impact of deployment errors on sensor worm propagation, which gives some practical guidelines for choosing appropriate system parameters to minimize the chance of worm propagation. Finally, our performance evaluation demonstrates the effectiveness of the proposed scheme in defending against sensor worm attacks. It also shows that our scheme greatly outperforms two other version assignment algorithms proposed in [27].

The remainder of this paper is organized as follows. First, related work is discussed in Section 2. Then, Section 3 studies the feasibility of launching sensor worm attacks. After that, Section 4 describes our graph coloring based sensor worm defense scheme. Performance evaluation is presented in Section 5. Finally, we summarize our work and discuss future work in Section 6.

2. RELATED WORK

Worm Defense Worm attacks in Internet [31] as well as buffer overflow vulnerabilities [28] have been extensively studied. Both proactive and reactive strategies are proposed to defend against worm attacks in Internet [11]. Also, Internet worm propagation is modeled and the impact of network topology on the size of the final infected population has been analyzed in [18, 14]. [33] proposes generic techniques for blocking buffer overflow attacks based on some inherent distinctions between exploit code and random data.

The related work we know on sensor worm mainly models the worm propagation. [13] models node compromise spread in wireless sensor networks using epidemic theory and identifies key factors determining potential outbreaks. However, the sensor deployment error and sensor compromise containment strategies are not considered in this work. Some techniques [24, 30] are also designed for sensor memory protection. These work together with [8] improve the defensive capabilities of individual sensors. It is necessary for us to enhance sensor network immunity under worm attack in a systematic way, since the defensive capability of individual sensors is limited. In parallel to our work, [20] illustrates that a mal-packet with only specially crafted data can exploit memory-related vulnerabilities and utilize existing application codes in a sensor to propagate itself without disrupting sensors' functionality.

Software Diversity Inspired by diversity, an important source of robustness in biological systems, software diversity in computer systems [17] as well as computer networking [27, 9] bears a lot of attention recently. A variety of randomization techniques [19] have been proposed to enhance the intrusion resistance of computer systems by increasing software complexity without degrading functionalities and performance. Diversification at the network level is achieved by applying different operating systems, critical software components [25] or communication protocols [26] on different machines of the network.

Similar work is conducted in preventing epidemics in the context of computer worms or viruses [10]. In [29], it is stated that selective immunization should be enforced according to the node's degree, i.e., nodes with high degree should be installed different softwares because they are more important in the network connectivity.

Graph Coloring Graph coloring (especially vertex coloring) [22], a famous problem in graph theory, ensures that there are no two adjacent nodes sharing the same color. So, its solution is a natural option for making globally optimal decision in software diversity. In the distributed coloring algorithms proposed in [27], the initially random assignment of nodes' colors will cause high communication overhead in the following color adjustment and negotiation with neighbors. Also, the algorithms may not converge to a few colors due to the high density of sensor networks, which in practice may result in a high cost for software implementations. Our sensor worm defense scheme has already considered all these factors as well as the sensor deployment errors, thus is well-tailored for sensor networks.

3. SENSOR WORM ATTACKS

In this section, we first introduce the memory architecture of sensors. Then, we illustrate the feasibility of launching sensor worms through experiments. At last, we model the propagation of sensor worms using a simple epidemic model.

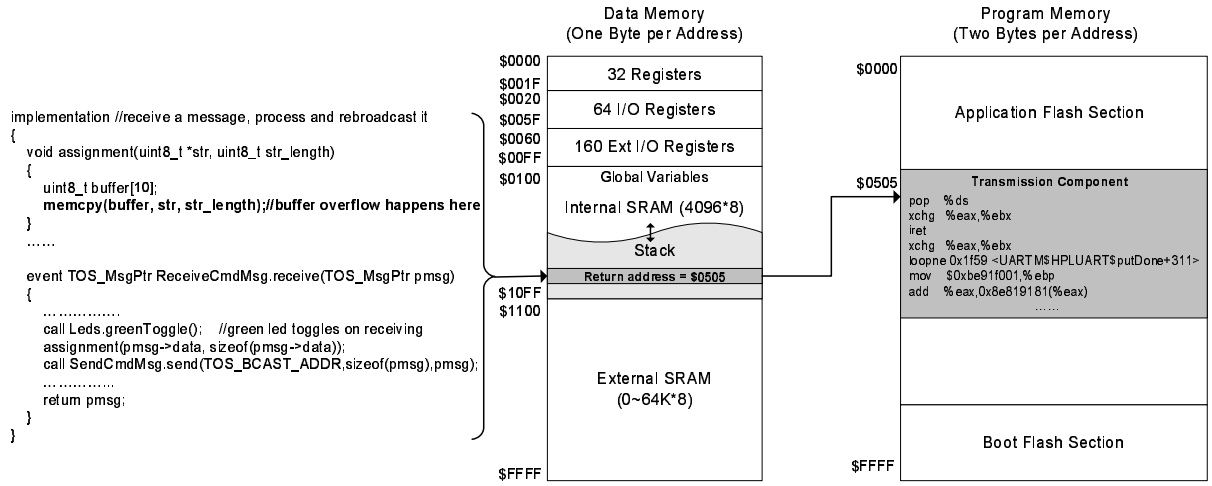


Figure 1: Buffer overflow experiment on memory structure of ATmega128.

3.1 The Memory Architecture of Sensors

Observing that most of the Internet worms exploit the buffer overflow vulnerabilities in software, we start by studying the buffer overflow problem in sensor nodes. Based on our initial investigation, we find that buffer overflow in sensors is memory architecture dependent. The 8 and 16-bit microcontrollers used in small embedded systems are designed in either of two memory architectures: the Von Neumann Architecture which uses a single physical memory for both program code and data, and the Harvard Architecture which uses separate memories for program and data.

Most of the buffer overflow attacks in the Internet target at the Von Neumann Architecture. By overflowing the boundary of a buffer in the stack or heap, a worm injects a piece of malicious code into a program. If it succeeds, it will cause the return address of a function to be overwritten; further, the instruction pointer will point to the location of the injected code and start to execute the malicious code. Therefore, if a sensor network consists of sensor nodes that use the Von Neumann architecture (e.g., microcontroller msp430 of Texas Instrument [6]), it is also vulnerable to such worm attacks if the program does not perform careful boundary checking.

The Harvard architecture, which is used by vendors like Atmel and Microchip, makes it hard (if not impossible) to inject malicious code, because the program memory and the data memory are separate (Figure 1). The data memory is not executable and typically different instructions are required to access the program and data memory. Mica2 motes [5] use the 8-bit microcontroller ATmega128 from Atmel [1]. The stack residing in the internal SRAM of the data memory is mainly used for storing temporary data, local variables and return addresses for subroutine calls and interrupts. The base of the stack (normally at 0x10FF) is defined in the RAMEND value of the include file (m128def.inc) for ATmega128 and the stack pointer is initialized to be 0x10FF too. This means when we push return address (two bytes) into the stack, we need two push instructions. The stack size is increased by two but the address in stack pointer will be decreased by two. Therefore, the maximum size of the stack equals to the size of the internal SRAM, which is 4KB.

The lowest addresses (around 0x0100) in the internal SRAM are normally allocated for the .bss section including uninitialized global or static variables (the size of this part is calculated for bytes in RAM during compilation) [4]. If the stack is overflowed by a large quantity of data, the stack size may grow to overflow the .bss section or even the register parts in the data memory. Thus, unexpected values will be set to the system variables and the registers. This kind of stack overflow, however, can only corrupt the current sensor and the attack will not propagate to other sensors automatically. Hence, during the experiment in the next section, we will focus on another kind of buffer overflow, which could cause the propagation of the attack in the network.

3.2 Trial Experiments on Sensor Worms

As a trial study, we are interested in stack-based buffer overflow, which causes the transfer of program flow by changing the return address of a function call. If a senor node has a routine that processes a received packet before relaying them, then by manipulating the content in the packet an attacker may change the return address of the function call to the beginning address of a transmission component in the program memory. Consequently, the sensor sends out the attack packet before it is corrupted. If the attack packet is supposed to reach all nodes (e.g., a broadcast packet), all the sensors will be affected.

To study the consequences of this type of buffer overflow, we performed some simple experiments on Mica2 motes. Suppose that the data structure of the message type contains a pointer to its data payload (string). Figure 1 shows the code in a sensor. When a sensor receives a message, it triggers the function Leds.greenToggle() to turn on/off the green LED. The sensor then calls a function assignment(), which copies the data payload to a buffer of size 10 without bound checking. Finally, it rebroadcasts the same packet to the network, and toggles a red LED after a successful transmission.

To start the message propagation process, one sensor broadcasts an attack packet, the data field of which is the beginning address of the transmission program in the sensor (which is 0x0505 in our case). In our experiments, we found that an extra 7-byte in the string is long enough to overflow the return address. Thus, by sending a packet with a 17-byte string in the data payload and setting every byte 0x05, the 2-byte return address of a receiving sensor will be changed to 0x0505.

When buffer overflow does not happen (the function assignment() is disabled), the red LED and green LED flash alternately on these two sensors, which means that two sensors operate normally and the packet is echoed between these two sensors. However, after the function assignment() is enabled, buffer overflow occurs and the return address is modified to point to the transmission component. Thus, both sensors become irresponsive after echoed the received packet once more (their red LED and green LED flash no more).

The above is a trial for constructing a sensor worm. Another possibility is to directly inject malicious code into the program memory through mechanisms such as over-the-air software distribution or network reprogramming [23]. Since the injected malicious code could be arbitrary as the attacker desires, it may bring more severe network-wide damages to the sensor network. We notice that sensor worm may adopt other vulnerabilities than buffer overflow to launch successful attacks. How they could be used is out of the scope of this paper. We may investigate more such possibilities in our future work.

3.3 The Modeling of Sensor Worm Attack

We use the classical simple epidemic model [10] to model the propagation of sensor worms when no defense is in place. Here sensors have two states: susceptible and infectious. The overall rate of new infections given by this model is:

$$\frac{dI(t)}{dt} = \frac{\beta I(t)(n - I(t))}{n}, \quad (1)$$

where $I(t)$ is the number of infectious nodes at time t, n is the total number of nodes in the network. β is worm's infection rate, which is the average number of probes an infectious node can send out to the population n during a unit time. Specifically, if we consider the topology of the sensor network as a graph, β is the average out-degree of a node. By solving this differential function, we can get the number of infectious sensors at any time t as:

$$I(t) = \frac{n}{1 + e^{-\beta t} C n}, \quad (2)$$

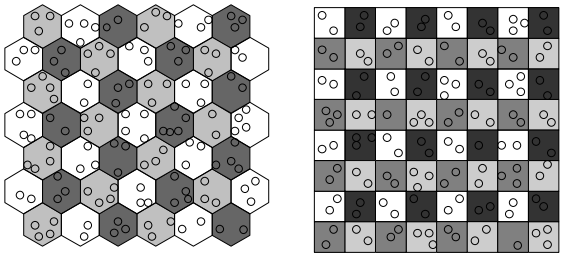
where C is a constant factor.

In Section 5.2.5, we will show the effectiveness of our proposed scheme in terms of infection fraction by comparing it with this no-defense case.

4. SENSOR WORM DEFENSE

Message source authentication can block sensor worms from outsider attackers, but it is not much helpful if sensor worms are injected by compromised sensors or base station. Our focus is to increase the survivability of sensor networks against sensor worms. In the spirit of *survivability through heterogeneity* philosophy [35], we will explore *software diversity* to combat sensor worms.

By software diversity, we should resort to multiple realizations (by different people) of the critical programs such as routing protocols and operating systems, then install each sensor node with one of the versions. We expect that different versions of the same functionality will not have the same vulnerability for the attacker to exploit, e.g., the beginning addresses of the transmission program in different implementations are different. On the other hand, due to the implementation cost it is unrealistic to assign every node a different version of a program. Therefore, the research challenge becomes: *given a limited number of versions of a program, how to optimize our global benefits in defending against sensor worms through software diversity?* Next we first present an efficient algorithm for version assignment (before deployment), followed by studying the impact of sensor deployment error (after deployment).



(a) 3-color case (b) 4-color case

Figure 2: Examples of color assignment.

4.1 Graph Construction

Graph coloring (especially vertex coloring) [22], a famous problem in graph theory, tries to assign colors to the vertices of a graph such that no two adjacent nodes in the graph share the same color. We may transform our problem into a graph coloring problem by first mapping a sensor network into a graph and mapping colors to the software versions(in the following, we use color and software version interchangeably), and then finding a solution to the corresponding graph coloring problem.

When mapping a sensor network into a graph, an intuitive solution is to map every sensor node into a vertex in the graph [27]. This, however, is not suitable for sensor networks, which usually have high density (e.g., a node may have 20 or more neighbors). A sensor network with high node density results in a high-density graph, making it infeasible to color the graph by using just a few colors. To address this problem, we will consider the geographic locations of sensors for mapping. Specifically, we will first partition the sensor field into small cells, then map every cell into a vertex and map the neighborship of two cells into an edge in the graph. In this model, cell is the unit of consideration and multiple sensors may locate in the same cell. Thus, when we assign a color to a cell, all the nodes in the cell will have the same color. Although in this case once a cell color is compromised more than one sensor will be corrupted, our focus is to quarantine the propagation of sensor worms so that large-scale node compromises will be prevented. Note that our scheme is suitable for applications in which a small fraction of node compromises may be tolerated.

4.2 Color Assignment

Before we describe the color assignment algorithm, we first introduce several concepts [27]. In a graph, an edge whose two endpoints having the same color is called a *defective edge*. Otherwise, it is called an *immune edge*. A *disconnected component* is a subgraph inside which all vertices are connected through defective edges whereas all its boundary edges are immune edges. Formally, suppose the set of cells in the network is denoted by V, the set of software is S and the number of software versions is $s = |S|$. Our purpose is to devise an assignment or mapping $S \mapsto V$, so as to (1) minimize the number of defective edges and (2) maximize the number of disconnected components. Indeed, these two goals are not orthogonal. This is because an increase in the

total number of disconnected components will reduce the size of each component as well as the number of defective edges.

According to Four Color Theorem, we only need a limited number of colors (e.g., no more than four) to implement our scheme. This means that our scheme is very practical because a very limited number of implementations for the critical function are sufficient to solve our problem. More specifically, our problem is to find out a color assignment to the graph so that $S(i) \neq S(j)$ if $(i,j) \in E$, given the topology of a graph $G = (V, E)$ and the available colors S. In the case that the number of available colors is larger than the optimally minimum one, the color assignment solutions are often not unique. We try to devise a color assignment algorithm that provides the flexibility of automatically outputting one of the viable solutions in an efficient way.

An intuitive solution is a greedy graph coloring algorithm. This algorithm as well as its improvement heuristics (e.g., first order vertices by decreasing or increasing degree) can find a solution fast, but their results are largely influenced by the order in the permutation of vertices. That is, in some circumstance, the greedy algorithm may have a conclusion that this graph cannot be colored by a certain number of colors, but once we change the vertex traversing order the greedy algorithm may successfully output a color assignment solution.

Our color assignment algorithm is based on backtracking [21]. Backtracking is a type of algorithm that is a refinement of the brute force search. In backtracking, many partial solutions can be eliminated without being explicitly examined, according to the properties of the problem [2]. In our case, once a vertex is assigned a color, then all its neighbors are refrained from being assigned the same color. It is a recursive procedure, independent of the vertex traversing order. In each recursion, the problem is turned to be a smaller problem with the same form. This process is repeated until every vertex is assigned a color or the procedure stops because of failing to assign a color to one of the vertices. The details of our backtracking color assignment algorithm is presented in Algorithm 1.

Let N be the total number of cells in the deployment area, which means that there are N vertices in the constructed graph. Clearly, time complexity of the brute force algorithm is $\Theta(s^N)$, because for each vertex from 0 to $N-1$ there are s choices. The backtracking procedure is a depth-first search on the solution space tree, which is a balanced tree with degree s and height N. Each internal node on the tree is to find the next available color by calling function availableColor(), with the time complexity of $O(sN)$. Our algorithm searches along a path from the root to the leaf in the tree, with the total number of internal nodes as N. Hence, the time complexity of the backtracking algorithm is $N \cdot O(sN) = O(sN^2)$. This polynomial time is much less than the exponential time of the brute force method.

We check the effectiveness of the above algorithm by applying it to the 3- and 4-color cases (the 2-color case is simply a column-based partition of the sensor field). For the 3-color case we may partition the sensor field using hexagons because its corresponding graph is 3-colorable. If we have four or more colors, then grid-based partition is applicable, because its corresponding abstract graph is 4-colorable. As shown in Figure 2, we construct one example of the solutions for each case, based on the outputs from Algorithm 1.

Algorithm 1 Backtracking Color Assignment Algorithm

Input: Adjacency matrix $G[0..N-1][0..N-1](N = |V|)$ of graph $G = (V, E)$, where $G[i][j] = 1$ if $(i,j) \in E$ and $G[i][j] = 0$ otherwise; Available colors are represented by integers $1..s$ where $s = |S|$;
Output: A color assignment solution represented by an array X, where $X[i](0 \leq i \leq N-1)$ is the color assigned to vertex i;
Procedure:

```
1: for i ← 0 to N − 1 do
2:    X[i] ← 0; {Initialize array X}
3: end for
4: backtracking(0); {Array X is updated here}
5:
6: void backtracking (int k)
7: loop
8:    availableColor(k);
9:    if (!X[k]) then
10:      break;{No new color for vertex k is available}
11:   end if
12:   if (k == N − 1) then
13:      printNodeColors();{A solution of array X is outputted}
14:      exit(0);
15:   else
16:      backtracking(k + 1);
17:   end if
18: end loop
19:
20: void availableColor (int k)
21: loop
22:   X[k] ← (X[k] + 1)%(s + 1);{Try the next color}
23:   if (!X[k]) then
24:      return;{All colors have been tried}
25:   end if
26:   for i ← 0 to N − 1 do
27:      if (G[k][i]&&(X[k] == X[i])) then
28:         break;{Vertices sharing an edge cannot have the same
           color}
29:      end if
30:   end for
31:   if (i == N) then
32:      return;{A new color is found}
33:   end if
34: end loop
```

We notice that both the 3-color and 4-color cases are optimal with minimum number of defective edges (which is zero) and maximum number of disconnected component (i.e., each cell is a disconnected component). To ease our presentation, our following discussion is mainly based on the graph topology of 4-color case in Figure 2(b).

The next research issue is: what should be the size of each cell? To answer this question, we need to take into account the transmission range of sensors, say R. Let the shortest distance between two cells of the same color be L. Clearly, we should make sure that $L > R$, so that there is no defective edge between them. As shown in Figure 2, L is actually the edge length of each cell, so the edge length of each cell should be larger than the transmission range R.

4.3 Sensor Node Deployment

So far we have been focused on color assignment in the ideal planning phase; we have not discussed how sensor nodes are deployed in a sensor field and whether a real deployment may cause issues. In real applications, there could be many ways to deploy sensor nodes in a field, subject to factors such as the requirement of the application and the safety of the deployment environment. In one extreme, we may be able to place every sensor node precisely in a

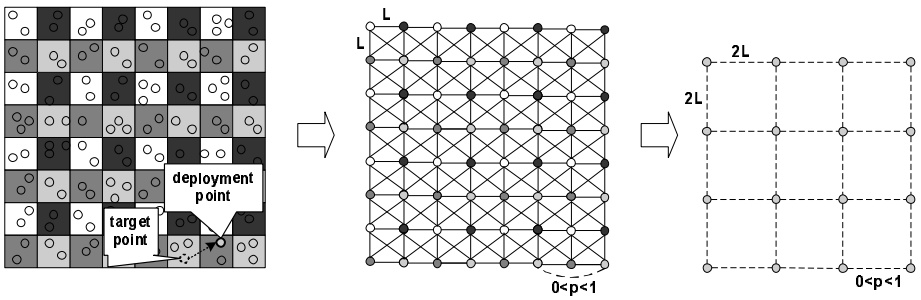


Figure 3: (Random) graph construction for 4 color based on original sensor deployment and color assignment.

planned location. If so, we may directly apply our color assignment algorithm. This however is unlikely for deploying a large-scale sensor network. In the other extreme, the real location of a sensor node is completely random in the sensor field. In this case, multiple sensors with different colors might be dropped in the same cell, rendering our algorithm falling back to a random coloring algorithm [27]. Inspired by the group-based sensor deployment model in [15, 16], we will study the case that the distance between the actual location of a sensor node and its targeted location follows a two-dimensional normal distribution (although some other probabilistic distributions may be considered as well). This is a reasonable model because in real applications sensors are normally prearranged according to their targeted locations and are then dropped out from a moving vehicle group by group.

More specifically, suppose that the deployment area ($X \times Y$) is divided into $h \times v$ cells and $(i,j)(1 \le i \le h, 1 \le j \le v)$ is the cell in row i, column j. If the target point of cell (i,j) is (x_i, y_j), we have the mean of sensor location distribution from this cell as $\mu = (x_i, y_j)$. The position (x,y) of sensors in this cell follows the pdf (probability density function) of a two-dimensional normal distribution $f_{i,j}(x,y)$:

$$\begin{aligned} f_{i,j}(x,y) &= f(x-x_i, y-y_j) \\ &= \frac{1}{2\pi\sigma^2} e^{-[(x-x_i)^2+(y-y_j)^2]/2\sigma^2}, \end{aligned} \quad (3)$$

where σ is the standard deviation of the distribution.

Clearly, the deployment error of sensor nodes will have some impacts on our objective. If two cells with the same color are separated by one cell of a different color according to our planning, there will be a chance that these two cells are connected due to deployment errors. In our setting, with some simplification(i.e., we do not consider cells in diagonal as neighboring in the random graph because they are farther away than cells neighboring in horizontal or vertical), we could construct a random graph for each color; that is, it consists of cells of the same color, as shown in Figure 3. There are four colors available, so four random graphs could be generated from the original sensor deployment and graph construction, one for each color. The probability p that an edge exists between two adjacent vertices in the random graph is related to the size of a cell (L), the deployment error (σ), the number of sensors in each cell (m), and the node transmission range (R).

We denote two cells separated by one cell of a different color as (i_1, j_1) and (i_2, j_2), with target points of (x_1, y_1) and (x_2, y_2) respectively. To derive p, we first consider the probability p_0 that one sensor N_1 from cell (i_1, j_1) is within the transmission range of another sensor N_2 from cell (i_2, j_2). Let us consider an infinitesimal rectangle $\mathrm{d}x \times \mathrm{d}y$ in the deployment area. The probability p_{n1} that sensor N_1 resides in this small area is:

$$\begin{aligned} p_{n1} &= f_{i_1,j_1}(x,y)\mathrm{d}x\mathrm{d}y \\ &= \frac{1}{2\pi\sigma^2} e^{-[(x-x_1)^2+(y-y_1)^2]/2\sigma^2}\mathrm{d}x\mathrm{d}y, \end{aligned} \quad (4)$$

because the probability density over this very small area can be treated as even.

Suppose the distance between node N_1 and the target point (x_2, y_2) of N_2 is d. Then, the probability that the sensor N_2 resides in the round area A centered at the location of sensor N_1 with radius R is different when $d \ge R$ and $d < R$. If $d \ge R$, as shown in Figure 4(a), the probability p_{n2} that sensor N_2 resides in the round area A is a double integration:

$$p_{n2} = \iint_A f_{i2,j2}(x,y)\mathrm{d}A.$$

We consider $\mathrm{d}A$ as the arc $\mathrm{d}A = \theta l \mathrm{d}l$. Because θ is a function of l, i.e., $\theta = 2arccos(\frac{l^2+d^2-R^2}{2ld})$, the above formula can be simplified to the integration:

$$\begin{aligned} p_{n2} &= \int_{d-R}^{d+R} f_{i2,j2}(x,y)\theta l \mathrm{d}l \\ &= \int_{d-R}^{d+R} 2arccos(\frac{l^2+d^2-R^2}{2ld}) f_{i2,j2}(x,y) l \mathrm{d}l. \end{aligned} \quad (5)$$

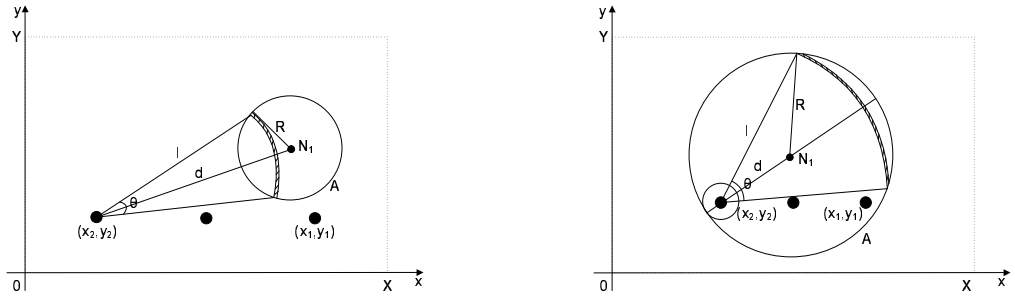
If $d < R$, as shown in Figure 4(b), the probability that sensor N_2 resides in the round area A equals to the same double integration, but now θ as a function of l has changed to:

$$\theta = \begin{cases} 2\pi, & \text{if } 0 \le l \le R-d; \\ 2arccos(\frac{l^2+d^2-R^2}{2ld}), & \text{if } R-d \le l \le R+d. \end{cases}$$

Hence, in this case the double integration can be simplified to the integration:

$$\begin{aligned} p_{n2} &= \int_0^{R-d} 2\pi l f_{i2,j2}(x,y)\mathrm{d}l + \\ &\quad \int_{R-d}^{R+d} 2arccos(\frac{l^2+d^2-R^2}{2ld}) f_{i2,j2}(x,y) l \mathrm{d}l \\ &= 1 - e^{-\frac{(R-d)^2}{2\sigma^2}} + \\ &\quad \int_{R-d}^{R+d} 2arccos(\frac{l^2+d^2-R^2}{2ld}) f_{i2,j2}(x,y) l \mathrm{d}l. \end{aligned} \quad (6)$$

Because node N_1 may appear at any place in the deployment area and the positions of two nodes from two different cells are two independent events, the probability p_0 that two



(a) $d \geq R$ (b) $d < R$

Figure 4: Derive p_0 that two nodes from neighboring cells with same color are within transmission range.

nodes from cells $(i1, j1)$ and $(i2, j2)$ are within each other's transmission range is:

$$p_0 = \int_{x=0}^{X}\int_{y=0}^{Y} p_{n2} \cdot f_{i1,j1}(x,y)\mathrm{d}x\mathrm{d}y = \int_{x=0}^{X}\int_{y=0}^{Y} \frac{p_{n2}}{2\pi\sigma^2} e^{-[(x-x_1)^2+(y-y_1)^2]/2\sigma^2}\mathrm{d}x\mathrm{d}y. \quad (7)$$

Next, we derive the probability p that two neighboring cells of the same color (in the random graph) are connected, based on p_0. The fact that two cells are connected means that at least one pair of nodes from these two cells respectively are within each other's transmission range. Let m be the number of nodes in one cell, then there are totally m^2 such kind of pairs. Therefore, the probability that two cells are connected is:

$$p = 1 - (1 - p_0)^{m^2}. \quad (8)$$

The above derivations will be validated by our simulation results in Section 5.2.1.

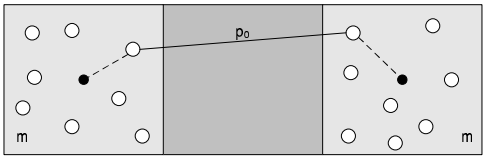


Figure 5: The derivation of probability p.

Based on the value of probability p, our next task is to check the impact of p on our goal of containing sensor worms. To address this problem, we resort to the percolation theory [7, 3]. Percolation theory deals with fluid flow (or any other similar process) in random media. In mathematics, percolation theory describes the behavior of connected clusters in a random graph. In our application, we will consider bond percolation (related to edges in graph) instead of site percolation (related to vertices in graph). Bond percolation means that the probability that one edge appears in the graph is $p(0 < p < 1)$. The problem is to find out whether there is a percolation cluster (i.e., infinitely extended connected cluster) going through the entire graph, which in our scenario means that sensor worm can successfully propagate throughout the entire network.

Clearly, as p increases, the average connected cluster size s_{av} also increases, because the probability of continuing a cluster by finding an adjacent connected edge becomes larger. By the size of a connected cluster, we mean the number of cells that are bound together by connected edges in the cluster. The cluster size distribution is typically expressed as a discrete function $\phi(\lambda)$, where $\phi(\lambda)$ is the number of clusters of size λ. Actually, when p is not close to 1, the probability of encountering a cluster of size λ is in the order of p^λ. For example, if $\lambda = 1$, then $\phi(\lambda) = p$, because if we choose an edge at random then the chance of it being connected is p. Similarly, if $\lambda = 2$, then $\phi(\lambda) = 6p^2$, and so on. More specifically, since the number of edges occurring in the clusters of size λ is proportional to $\lambda\phi(\lambda)$, the weighted mean of cluster sizes is given by:

$$s_{av} = \frac{\sum_{\lambda=1}^{n_r} \lambda^2\phi(\lambda)}{\sum_{\lambda=1}^{n_r} \lambda\phi(\lambda)}, \quad (9)$$

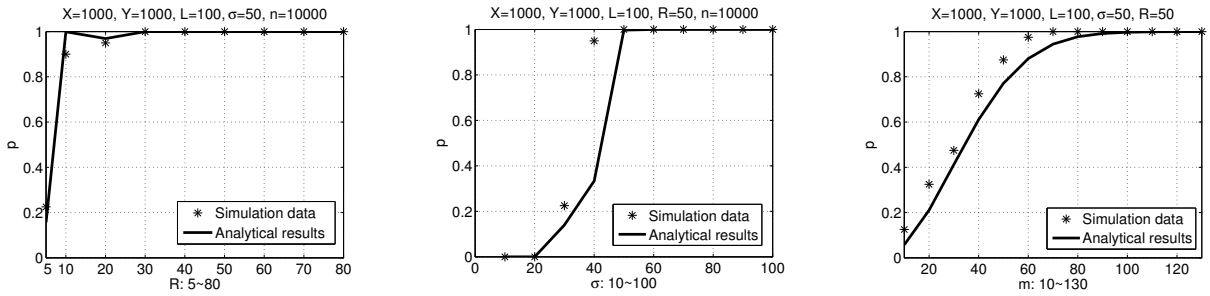
where $n_r (n_r \leq n)$ is the total number of nodes with the same color in the random graph. $n_r = n$ when there is only one available color. From the above formula, we know that the average cluster size s_{av} approaches n_r when p is close to a critical value p_c. For instance, $p_c \approx 0.65$ for hexagons (Figure 2(a)), and $p_c \approx 0.5$ for the square lattice in two dimension (Figure 2(b)). When $p < p_c$ the probability that a percolation cluster exists is close to 0, but when p is increased to around or above p_c the probability that a percolation cluster exists rapidly approaches 1. This is verified by our following simulation results in Section 5.2.2.

Hence, we should choose p_0 as close to 0 as possible so that p becomes small and lower than p_c (according to Eq.(8)). In real applications, normally the transmission range and deployment error level are fixed (although in Section 5.2.4 we will check the situation that the attacker can increase the transmission range of compromised nodes to gain more from the attack), so intuitively we should make the cell size L as large as we can to obtain small p_0 and p (when other parameters such as σ and R are fixed, if L is larger, then p_0 is smaller so that p is also smaller). However, we notice that it is not always better if L is larger, because in that case there will be more sensors in each cell. Suppose the length of the edge of the deployment area is $|X| = |Y| = \delta$. Then, the number of nodes in each cell is:

$$m = \frac{n}{(\delta/L)^2} = \frac{nL^2}{\delta^2},$$

assuming that each cell has the same number of sensors. Clearly, m increases with L, when δ and n are fixed. One extreme case is that L equals to δ. Then, all the sensors in the whole deployment area have the same color. At this time, the worm can easily propagate over the whole area.

The above formulation provides us a way to pick up the parameter L for cell size. Suppose the maximum size of connected clusters exists in the random graph is s_{max}. Then, the number of sensor nodes that can be compromised at one time by the worm attack is ms_{max}, because there are m sensors in each cell. Our objective is to keep this number



(a) p as a function of R (b) p as a function of σ (c) p as a function of m

Figure 6: Probability p of two neighboring cells with the same color connected, as a function of R, σ, m.

as small as possible. To achieve this, if L is larger, then s_{max} is smaller because p is smaller, but at the same time, m becomes larger. Therefore, we can have a minimum of ms_{max} under a certain value of L. We will use simulations to find out this optimal value of L in Section 5.2.3.

4.4 The Case of Loading Multiple Colors

Above we assume that each sensor can only take one color, which is the color of its targeted cell. We notice that if the program memory of a sensor node has extra space to hold multiple versions of code (OSes or other programs) and it allows dynamic transition of codes, it may be preloaded with multiple versions altogether. In addition, we can preload each sensor node with the optimal planning map, which marks the color of each cell corresponding to the case of no deployment error. After its deployment, a sensor node first figures out the cell it falls in based on an attack-resilient localization scheme [12], and then sets its color according to the planning map. In this case, deployment error will have no impact on the scheme.

In [27], O'Donnell and Sethu proposed several distributed coloring algorithms for networks consisting of machines that can be loaded with multiple colors. In their algorithms, every machine is mapped to a vertex. After deployment, a machine may randomly select a color out of a set of available colors (called *random coloring*) or flips its color based on its neighbor's colors (called *color flipping*). These algorithms however are not well tailored for sensor networks that feature high network connectivity; moreover, a relatively high communication cost is needed for a node to negotiate colors with its neighbors in a sensor network. Our scheme does not have these limitations. We will quantitatively compare our scheme with theirs in the next section.

5. PERFORMANCE EVALUATION

In this section, we will validate our analytical model and show the performance of our scheme under different parameter settings. The results will also be compared with those of randomized coloring and color flipping schemes in [27].

5.1 Simulation Setup

In the simulation, the deployment area of X×Y=1,000m× 1,000m is divided into different sizes of cells with edge lengths from 25m to 250m respectively. The target points of each cell is the center. The total number of sensors in the entire network changes from 1,000 to 64,000. The transmission range varies from 5m to 90m. The standard deviation of the two-dimensional normal distribution is in the range of 10 to 100. The number of available colors is either 4 or 5.

In our simulation, by default, there are totally 10,000 sensors distributed over the deployment area and each cell has the edge length of 100m. On average, the node density over the entire area is 100 sensors per cell (100m×100m), i.e., one sensor per 10m×10m area. We may adjust the distribution of sensors by changing σ. For example, under the same cell size, when the target points are 2σ=100m apart (i.e., $\sigma = 50$), the overall probability distribution function is close to uniform. Note that each point in the following figures is averaged over 100 rounds.

5.2 Simulation results

5.2.1 Probability p of Two Cells Being Connected

We check how p changes with R, σ and m through both analytical results and simulations. In the simulation, p is computed as the number of edges that are connected divided by the total number of edges in the random graph (of the same color). As shown in Figure 6, the simulation results match our analytical results well.

In Figure 6(a), with R being changed to above 10, p is increased rapidly from 0.2 to 1 under $\sigma = 50, L = 100$ and $n = 10,000$. In Figure 6(b), when σ is increased to above 40, p also quickly increases from 0 to 1. In Figure 6(c), when the number of sensors in each cell increases, p is increased to 1 gradually.

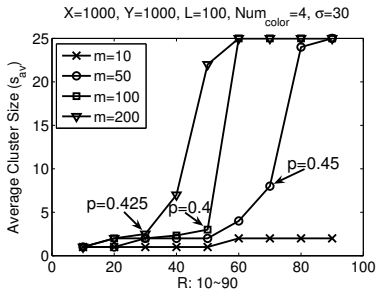
5.2.2 The Average Cluster Size

Figure 7 shows that the average cluster size changes as a function of R and σ under different m. When L is 100, there are totally 10×10 cells in the network. Since there are 4 colors, each random graph of one color has 25 vertices and 40 edges. When R or σ is larger than a threshold, the average cluster sizes are increased rapidly to cover the entire random graph. In reality, this means that we should keep R and σ to be relatively small values, e.g., much less than 40, so that sensor worms cannot propagate over the deployment area.

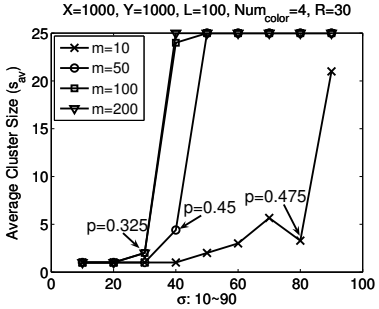
We also check the values of p at the turning points in each line of Figure 7. p is normally between 0.4 and 0.5 at those points, which means the critical value p_c is around $0.4 \sim 0.5$ in our scheme. This confirms our derivations based on percolation theory and Eq.(9) in Section 4.3.

5.2.3 Maximum Number of Node Compromises

Next, we study the maximum number (s'_{max}) of node compromises that could happen at one time. In the simulation, s'_{max} is calculated by multiplying the maximum size of connected cluster with the number of sensors at each cell ($s_{max} \times m$). As shown in Figure 8, s'_{max} changes as a function of cell size L and the total number of sensors n.



(a) Average cluster size vs R



(b) Average cluster size vs σ

Figure 7: The average cluster size changes as a function of R and σ.

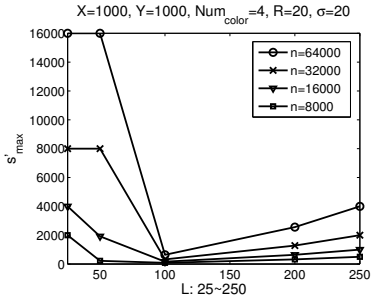


Figure 8: s'_{max} changes as a function of cell size L under different n.

Under a certain number n, when L is larger, the probability p that two cells with the same color are connected becomes smaller, so the maximum size of connected cluster (s_{max}) is also reduced. At $R = 20$ and $\sigma = 20$, it is decreased to the size of 1 (i.e., containing only one cell) quickly until L reaches 100. After this, s_{max} remains 1, but the number of sensors in each cell is increased with L. Thus, s'_{max} starts to increase (slowly). Therefore, we achieve the minimum s'_{max} when $L = 100$ under this condition. On the other hand, if we decrease n, the above trend is the same, but the absolute values of s'_{max} become smaller.

5.2.4 Comparison with Previous Schemes

We compare the performance of our scheme with the previous schemes, random coloring and color flipping [27], in terms of s'_{max}. In random coloring, n (changing from 1,000 to 10,000) sensors are randomly deployed over an area of 1,000m×1,000m, each of which has a random color. Under the same setting, in color flipping every sensor changes its color if necessary according to its neighbors' colors, so as to reduce the defective edges. Since there is no concept of cell in random coloring and color flipping, s'_{max} is the maximum number of connected sensors with the same color.

We first examine s'_{max} as a function of the total number of sensors n. The simulation results are presented in Figure 9(a). In both these two schemes, s'_{max} rapidly approaches the maximum number of sensors with the same color (2,500 in 4-color case and 2,000 in 5-color case), when the total number of sensors increases. On the other hand, more available colors help only when n is relatively small, because in the figure only when $n < 6,000$ s'_{max} is smaller with 5 colors than with 4 colors. In our scheme, there are totally $\frac{n}{100}$ sensors in each cell when $L = 100$. s'_{max} does not change much under appropriate parameters such as a small $\sigma(= 30)$. Even when $n = 10,000$, s'_{max} is as low as 200.

We also check s'_{max} as a function of the transmission range R. Although sensors normally have a fixed transmission range, transmission range R in our simulation varies because we notice that after the attacker compromises one sensor he may increase the transmission range of the compromised node by enhancing its RF power level or antenna configuration to maximize his gain from the attack. The simulation results are presented in Figure 9(b). Similarly, in both random coloring and color flipping, s'_{max} is increased rapidly to close to the maximum number of sensors with the same color in the network, even when R is as small as 30 under $n = 10,000$. Compared with Figure 9(a), more available colors (from 4 to 5) is even useless, because the 4-color and 5-color lines are almost overlapped together. In our scheme, there are totally 100 sensors in each cell when $L = 100$ and $n = 10,000$. If R is small, sensor worm will not propagate outside one single cell, so the maximum number of sensors that could be compromised at one time is 100, which may be slightly higher than those in the other two schemes at the beginning. However, our scheme significantly outperforms the other two as R becomes larger, because s'_{max} does not change much under appropriate parameters.

The above results indicate that our scheme with proper parameters can effectively prevent the propagation of sensor worms, compared with the previous two schemes.

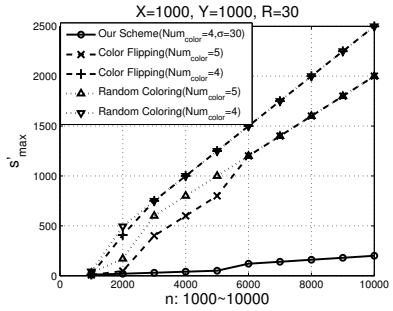
5.2.5 Effectiveness of Sensor Worm Containment

At last, we simulate the infection rate $I(t)$ in our scheme and compare the results with those in the simple epidemic model and the random coloring scheme. We check the percentage of infectious nodes as a function of time units for probing rate $\beta = 4$ and $\beta = 6$ (although in reality sensor worm has broadcast nature in probing, we check these two cases merely for comparison reasons). As shown in Figure 10, $I(t)$ in our scheme increases much slower than those in the original theoretical model and the random coloring scheme. The comparison results indicate that our scheme can substantially reduce the possibility that sensor worm propagates in the network.

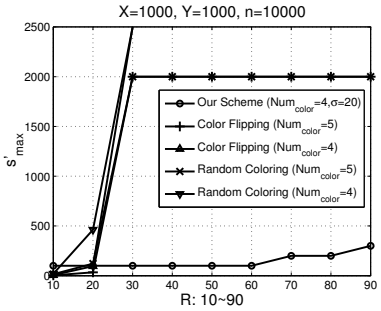
Through simulations, we find that the simple epidemic model can roughly describe the propagation rate of sensor worm when all the sensors have the same color in the network. In this case, sensor worm can propagate throughout the entire network of 10,000 sensor nodes (100% coverage) within 2 or 3 time units. Moreover, when the probing rate β becomes larger (e.g., increased from 4 to 6), i.e., an infectious node sends more probes to the population at one time, the number of infectious nodes is increased even faster. Fortunately, having multiple colors (in random coloring) can slow down this process in a large degree. The percentage of infectious nodes in random coloring is eventually increased to 25% when the number of available colors is 4 at the 10th time unit. Compared with random coloring, our scheme can further improve the defensive capability against sensor worm by quarantining sensor worm within as few cells as possible. As a result, at the 10th time unit, there are only 3.2% infectious nodes when $\beta = 4$ and 4.76% infectious nodes when $\beta = 6$, under the condition that $\sigma = R = 30$ and $L = 100$.

6. CONCLUSION

In this paper, we not only illustrate the feasibility of launching worm attacks in sensor network, but also propose a concrete defense scheme based on the idea of software diversity.



(a) s'_{max} as a function of n



(b) s'_{max} as a function of R

Figure 9: Comparing s'_{max} of our scheme with others.

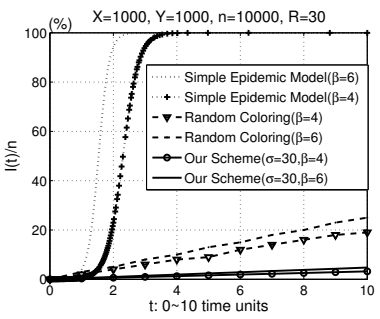


Figure 10: Comparing $I(t)/n$.

We show by assigning each sensor an appropriate version of software among a few versions, we can significantly increase the survivability of sensor networks under worm attacks. We also analyze the impact of sensor deployment error on the capability of our scheme, which provides a guidance for our selection of system parameters. In the future, we may investigate the impact of different cell shapes, such as hexagon, triangle, and other polygons.

Acknowledgments We are grateful to Piotr Berman for inspiring discussion on graph coloring. We also thank the anonymous reviewers for their helpful comments.

7. REFERENCES

[1] *ATmega128(L)*. http://www.atmel.com/dyn/resources/prod-documents/doc2467.pdf.

[2] Backtracking. http://en.wikipedia.org/wiki/Backtracking.

[3] Introduction to Percolation Theory. http://garnet.berkeley.edu/ jqwu/paper1/paper1.html.

[4] *Memory Sections in Related Pages*. http://hubbard.engr.scu.edu/embedded/avr/doc/avr-libc/avr-libc-user-manual/.

[5] *Mica Motes*. Crossbow Technology, Inc. http://www.xbow.com.

[6] *MSP430 Microcontrollers*. Texas Instrument. http://www.ti.com/.

[7] Percolation theory. http://en.wikipedia.org/wiki/Percolation-theory.

[8] A. Alarifi and W. Du. Diversify sensor nodes to improve resilience against node compromise. In *SASN'06*, 2006.

[9] M. G. Bailey. Malware resistant networking using system diversity. In *SIGITE '05*.

[10] N. Bailey. *The mathematical theory of infectious diseases and its applications*. Hafner Press, New York, 1975.

[11] D. Brumley, L.-H. Liu, P. Poosankam, and D. Song. Design space and analysis of worm defense strategies. In *ASIACCS*, 2006.

[12] S. Capkun and J. Hubaux. Secure positioning in sensor networks. Technical Report Technical Report EPFL/IC/200444, 2004.

[13] P. De, Y. Liu, and S. K. Das. Modeling node compromise spread in wireless sensor networks using epidemic theory. In *WOWMOM '06*.

[14] M. Draief, A. Ganesh, and L. Massoulie. Thresholds for virus spread on networks. In *ValueTools'06*.

[15] W. Du, J. Deng, Y. S. Han, S. Chen, and P. Varshney. A key management scheme for wireless sensor networks using deployment knowledge. In *IEEE INFOCOM*, 2004.

[16] W. Du, J. Deng, Y. S. Han, and P. Varshney. A key predistribution scheme for sensor networks using deployment knowledge. *IEEE Transactions on Dependable and Secure Computing*, 2006.

[17] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *HOTOS '97*.

[18] A. Ganesh, L. Massoulie, and D. Towsley. The effect of network topology on the spread of epidemics. In *Infocom*, 2005.

[19] G.S.Kc, A.D.Keromytis, and V.Prevelakis. Countering code injection attacks with instruction set randomization. In *CCS*, 2003.

[20] Q. Gu and R. Noorani. Towards self-propagate mal-packets in sensor networks. In *ACM WiSec'08*.

[21] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms/C++*. Computer Science Press, 1996.

[22] T. R. Jensen. *Graph Coloring Problems*. Wiley, 1995.

[23] S. Kulkarni and L. Wang. MNP: Multihop network programming for sensor networks. *Proc. of International Conference on Distributed Computing Systems*, 2005.

[24] R. Kumar, E. Kohler, and M. Srivastava. Harbor: Software-based memory protection for sensor nodes. In *IPSN*, 2007.

[25] M.C.Mont, A.Baldwin, Y.Beres, K.Harrison, M.Sadler, and S.Shiu. Towards diversity of cots software applications: Reducing risks of widespread faults and attacks. In *Technical Report HPL-2002-178*, 2002.

[26] N.Roux, J-S.Pegon, and M.Subbarao. Cost adaptive mechanism to provide network diversity for manet reactive routing protocols. In *MILCOM*, 2000.

[27] A. J. O'Donnell and H. Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *CCS '04*.

[28] A. One. Smashing the stack for fun and profit. *Phrack 49*. http://www.phrack.org/show.php?p=49a=14.

[29] R. Pastor-Satorras and A. Vespignani. *Epidemics and immunization in scale-free networks*, chapter Handbook of graphs and networks: from the genome to the Internet. 2002.

[30] J. Regehr, N. Cooprider, W. Archer, and E. Eide. Memory safety and untrusted extensions for tinyos. Technical Report UUCS-06-007, University of Utah, 2006.

[31] S. Staniford, V. Paxson, and N. Weaver. How to own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[32] M. Vojnovic and A. Ganesh. On the race of worms, alerts and patches. In *ACM WORM 2005*.

[33] X. Wang, C. Pan, P. Liu, and S. Zhu. SigFree: A Signature-free Buffer Overflow Attack Blocker. In *USENIX Secucrity'06*, July 2006.

[34] Y. Yang, X. Wang, S. Zhu, and G. Cao. SDAP: A secure hop-by-hop data aggregation protocol for sensor networks. In *Mobihoc*, 2006.

[35] Y. Zhang, H. Vin, L. Alvisi, W. Lee, and S. K. Dao. Heterogeneous networking: A new survivability paradigm. In *NSPW'01*.