Efficient Code Diversification for Network Reprogramming in Sensor Networks

Qijun Gu
Department of Computer Science
Texas State University-San Marcos
601 University Drive, San Marcos, TX 78666
qg11@txstate.edu

ABSTRACT
As sensors in a network are mostly homogeneous in software and hardware, a captured sensor can easily expose its code and data to attackers and further threaten the whole network. To increase the survivability of a sensor network, code diversification has been shown to be an effective solution. However, disseminating many diversified code images is very costly in current network reprogramming systems, as it does not take advantage of the epidemic propagation of network reprogramming. New mechanisms need to be studied for integrating code diversification with network reprogramming. This paper proposes an efficient code diversification scheme for network reprogramming in sensor networks. The scheme uses Deluge to disseminate code images of sensor applications that carry diversification information and allows sensors to randomize the layout of their executables. Such diversification can defeat a wide range of attacks that exploit the knowledge of code layout, as no sensors have the same code layout in their executables. Except the cost determined by the code size, the computational overhead of diversification in sensors can be reduced to 60%, while sacrificing only 10% of security.

Categories and Subject Descriptors
D.4.6 [Operating Systems]: Security and Protection

General Terms
Security

1. INTRODUCTION
Sensor networks are often less attended. After sensors are deployed in field, they are prone to physical capture. With a captured sensor, attackers can reverse engineer code in the sensor. Because sensor networks have many homogeneous devices, the code obtained from a captured sensor can help attackers compromise other remote sensors. Such attack techniques in sensor networks have been identified in recent research that leads to the creation of remote reprogramming gadgets [7] and worm-like malware [9].

To increase the survivability of sensor networks, one of the defense ideas is to introduce diversity into a sensor’s code and data. It has been shown that such an idea is effective in raising the difficulty of successful capture attacks [2] and countering the propagation of exploiting packets [19]. Software diversity has been intensively studied in the past. Various schemes have been proposed to generate different versions of the software with the same functions and then load each sensor with one of the version.

However, such offline diversification incurs much difficulty in managing and updating code in sensors. At the same time, network reprogramming [11, 13] has emerged as a major technique that can conveniently manage, update and patch the code of sensors in field. It can quickly disseminate a new code image to sensors in a network via epidemic propagation. New mechanisms of code diversification need to be studied for network reprogramming, because disseminating many diversified code images is very costly as it does not take advantage of the epidemic propagation in current network reprogramming schemes.

In this paper, a new code dissemination scheme is developed for diversifying code in sensors. The proposed approach is similar to run-time randomization [18] used in regular computers that randomizes functions, stacks, heaps and etc. When a program is to be loaded into memory for execution in a regular computer, the loader can randomly set up the layout of the code and the data in memory, such that different computers have different run-time code even if the running code comes from the same program. However, the current network reprogramming approaches do not support such randomization. They simply send a code image to the sensors, which will load the program in the code image to execute.

The process of the proposed code diversification scheme in a sensor network consists of three steps. First, a new code image is generated from a real executable ahead of code dissemination. The image contains the transformed code blocks and the information that helps glue the code blocks in sensors for diversifying code. Then, the code image is disseminated to sensors using existing network reprogramming schemes, and stored in their flash memories. Finally, when a sensor needs to use the program carried in the code image, the sensor builds an executable from the stored code image using its internal random data. Therefore, each sensor can independently randomize code and the whole network is diversified.

The proposed scheme does not change the underlying code dissemination protocol and thus inherits its advantages. Nevertheless, the scheme does not disseminate the original program. Rather, it utilizes a code image that contains sufficient information for sensors to rebuild diversified executables. Because sensors are limited in computational and energy resources, the key of the scheme is to have an efficient information structure in the code image to support randomizing executables.

The proposed code diversification scheme is the first to explore the mechanism of diversifying sensor software with the current network reprogramming system Deluge. The contribution of the pro-
posed scheme includes three aspects. i) The proposed scheme is application independent. It allows sensors to diversify the complete code image to provide system-wide protection, including Deluge and the proposed scheme themselves. ii) A diversification graph is proposed to capture the fundamental features of code randomization in sensor applications. The graph shows the relation between the security of randomized code and the overhead of the proposed scheme. iii) A quantitative measurement is developed to quantify the security achieved in randomization. The quantification of security helps develop strategies for constructing the diversification graph with different overhead and security requirements.

The rest of the paper is organized as follows. Section 2 gives the background on network reprogramming in sensor networks. Section 3 presents the proposed code diversification scheme and discusses the key issues. Section 4 describes the implementation and analyzes the overhead and security of the scheme. Section 5 summarizes the related work. Section 6 concludes the paper.

2. NETWORK REPROGRAMMING

Several network reprogramming schemes have been developed for wireless sensor networks in the past. Some [11, 13] support full image reprogramming that completely replaces an old program with a new program, while others [12, 17] support incremental image reprogramming that replaces only the changed part of a program. Since this paper targets diversifying the complete code image, the full image reprogramming approach is selected. This paper builds the code diversification upon Deluge [11], which is a full image reprogramming mechanism widely used in research and development of sensor networks and available in TinyOS [1].

The dissemination of a code image in Deluge starts from a base station and follows the epidemic propagation model. The base station sends the code image to its nearby sensors, which further send the image to their nearby sensors. Because a code image usually has thousands of bytes, the code image is divided into pages. Each page is made of 48 packets, and each packet (the base unit in transmission) has 23 bytes. Sensors recover the complete code image after receiving all packets and pages. The code image is then stored into sensors’ external flash memories. Figure 1(a) shows how a code image is stored in a volume of flash memory. The code image is composed of three parts. The first part is the Deluge header, which contains the meta information of the code image. The second part is the CRCs of code pages. Each CRC is computed for one page of the code. The last part is the actual executable.

The procedure of reprogramming and rebooting takes a few steps under the control of the bootloader TOSBoot. When a sensor receives the command to reprogram itself with a code image stored in its flash memory, it transfers the control to the bootloader. The bootloader then verifies if the Deluge header is correct against its own crc and if all code pages are correct against the CRCs of code pages. If either is incorrect, the bootloader reboots the sensor with the current running program. If both pass, the bootloader copies the executable stored in the flash volume to the program memory of the sensor. Finally, the bootloader reboots the sensor to execute the new program.

The process of network reprogramming in Deluge shows that the underlying code dissemination protocol does not care whether the code image contains a ready-to-execute program or not. Rather, the dissemination itself treats the code image simply as a big chunk of data. All sensors store such data in their flash memories upon receiving it. Furthermore, internally, the TOSBoot simply overwrites the current running program with a selected program stored in a flash volume. Hence, it is possible to add a hook into Deluge such that (i) a code image carrying diversification information can be disseminated as data to all sensors and then (ii) the code image can be transformed to an executable program by sensors themselves before TOSBoot is invoked.

The paper will discuss how to generate such a code image with additional information for code diversification and how to transform the code image into a diversified executable. The paper will also discuss and analyze the key to reducing overhead of code diversification in network reprogramming.

3. CODE DIVERSIFICATION

3.1 Attack Models and Assumptions

This paper does not consider attacks that simply capture nearby sensors. Instead, it tackles attacks that send malicious packets to exploit vulnerabilities in remote sensors. We assume that attackers can obtain the binary image from a captured sensor, have sufficient time to reverse engineer the code, can quickly develop exploiting packets offline, and then launch attacks. Researchers have found techniques that use exploiting packets to redirect the control flow to achieve these attacks. Such attacks help attackers obtain control over remote sensors that are not in their nearby areas. Such attacks can effectively threaten a network of hundreds of sensors.

Diversifying code in all sensors can defeat such attacks, because obtaining the code of one sensor will not give more advantage to attackers to compromise another sensor. Among all the code diversity approaches, randomizing code layout [4] is suitable to sensors. By randomizing the layout of the code in each sensor’s program memory, an attacking packet that can exploit the code of one sensor will not be able to exploit another sensor. This approach is also efficient in that only the layout of code needs to be randomized in sensors. Such layout information can be carried and utilized in the process of network reprogramming. Therefore, the proposed code diversification scheme in this paper will be built upon code layout randomization.

3.2 Overview of Code Diversification

The proposed code diversification scheme (the CD scheme in brief) involves two phases. In the first phase, the application developers create a code image with diversification information. This image is then disseminated to all sensors in a network from the base station. The dissemination of the code image follows the protocol used in the original Deluge. In the second phase, each sensor utilizes the diversification information in the received code image to recover a diversified executable and then reboots to execute the diversified program.

As the base station works on the code image ahead of dissemination, it has sufficient time to manipulate the code image. Hence, the key to an efficient code diversification for networking reprogramming is how a base station can construct a data structure to contain sufficient diversification information such that sensors can restore...
discussed in detail in Section 3.4. The relation between security and overhead will be quasi-executable and a the executable in the new code image is replaced with a

Figure 2(a). First, the source code of a sensor application is com-

tion.

This phase is carried out in the base station as illustrated i n

a diversified executable with minimal effort.

Phase 1: creating a code image with diversification information. This phase is carried out in the base station as illustrated in Figure 2(a). First, the source code of a sensor application is compiled into an executable. Then, the executable is utilized to analyze the control flow graph (CFG) of the application. A diversification graph (DG, which will be discussed in detail in Section 3.3) will be built upon the CFG. There are quite some tools to analyze the CFG of a program at the source code level. However, this research built tools that directly analyze the CFG and build the DG atop the executable. In this way, the actual sequence of code execution will not be changed and the optimization of the application will be untouched in the diversified executable. After a DG is obtained, the CD scheme restructures the DG according to security and overhead requirements. The relation between security and overhead will be discussed in detail in Section 3.4.

Figure 1(b) shows the structure of the code image with diversification information. Compared with the code image in Figure 1(a), the executable in the new code image is replaced with a CD header and a quasi-executable. The deluge header and the CRCs are recomputed based on the new code image. The CD header includes three components. First is a block of meta data containing auxiliary information for rebuilding the executable. The second component is a node table, in which each entry points to the location of a control node in the quasi-executable. The third component is an ETA table, in which each entry gives the address information of an ETA (refer to Section 3.3). The quasi-executable is a transformation of the original executable. The ETAs in the original executable are converted to indexes that point to the corresponding entries in the ETA table. All code other than the ETAs are not used in diversification and thus kept untouched.

Phase 2: recovering a diversified executable. The process of recovering a diversified executable in sensors is shown in Figure 2(b). The recovery utilizes a temporary memory to store information for diversification. It first copies the node table in the code image to the temporary memory, and then randomizes the code layout and stores the randomization information in the node table. After randomization, each entry in the node table contains the starting address of a control node and the index of the next control node in the randomized layout. Then, the recovery procedure copies the ETA table to the temporary memory without any change.

With the randomized code layout information ready in the temporary memory, the recovery procedure can start copying the quasi-

executable code into a dedicated flash volume, which is always used for storing the final diversified executable. The code image in the original flash volume will neither be overwritten, nor be used for rebooting. Hence, the original code image can be used by Del-

Figure 3: ETA and ITA in a DG

gue for dissemination. When copying the code to the dedicated flash volume, the recovery routine further computes the Deluge header for the executable. The new Deluge header is needed for the TOSBoot program to load the executable into the program memory and restart the sensor. The Deluge header of the code image in the original flash volume cannot be used for rebooting, because it is not for the diversified executable. Finally, the recovery process hands control over to the TOSBoot program to restart the sensor.

3.3 Diversification Graph

The key to building the diversification information is to find the data in code that can be randomized for diversification. This paper builds a diversification graph (DG) using such data and utilizes the DG to analyze security and overhead of the CD scheme.

Normally, a program can be depicted as a control flow graph (CFG). The graph contains a set of control nodes and transitional edges among the nodes. A control node is a block of instructions that are executed sequentially and stored in a continuous address space. A transitional edge between two nodes is made by various kinds of jump, call or branch instructions. However, a CFG does not contain sufficient and accurate information for diversification, as a transitional edge shows only one kind of relation between two nodes. The CD scheme builds a DG, which uses the same set of control nodes of the CFG, but creates edges among nodes based on two types of constants in code.

• An external target address (ETA) is a constant in a control node, which points to a code address in another control node.

• An internal target address (ITA) is a constant in a control node that points to a code address within the same control node.

ITAs and ETAs include not only the operands of call and jump instructions but also various kinds of constant data embedded in code. They are categorized according to their usage in diversification. Figure 3 shows an example of two control nodes $F_A$ and $F_B$. An ITA in a DG does not produce an edge between two nodes. Whereas, if a node has an ETA pointing to another node, a directional edge will be created between them.

As sensors decide the address of each control node in diversification, the base station cannot set ITAs and ETAs in the code image to be disseminated. Hence, auxiliary information must be provided for sensors to know what ITAs and ETAs are and how to recom-

pute them in diversification. Figure 3 shows how the CD scheme changes the ETAs and the ITAs in the original code so that they can be correctly recovered in diversification.

Because a function normally uses less than a half of the total program address space, the most significant bit (MSB) of an ITA can
be marked as 0 and the rest of the address bits can be set as an offset of the original ITA in the control node. During diversification, if the starting address of the control node is known, the ITA can be recomputed as the sum of the starting address and the offset. In contrast, for an ETA, its MSB is marked as 1 and the rest of its bits are set as an index pointing to an entry in the ETA table. The entry contains the ID of the target control node and the offset of the ETA in that control node. Therefore, when the starting address of the target control node is known during diversification, the ETA can be recomputed as the sum of the starting address of the target control node and the offset in that control node.

### 3.4 Randomization and Security Analysis

As the control nodes can be stored in any location in program memory, the layout of a program can be randomized inside each sensor. Any randomization algorithm can be used to randomize the code layout. This paper simply shuffles the nodes in the node table with an internal random number generator in each sensor. Even though the code layout is randomized, it cannot guarantee that the code within a control node cannot be exploited, since the internal of the control node is not randomized. For example, an attacking packet, using relative destination address, may redirect the control flow to another address within the same control node to exploit the code at that address. Hence, the security brought by the code layout randomization is determined by (i) the possibility that an exploitable vulnerability exists within a control node and (ii) the possibility that a piece of code can be utilized in exploitation.

Assume a program consists of \( N \) control nodes, and each node has \( c_i \) bytes of instructions, for \( 0 \leq i < N \). Using the linear model of vulnerability in software, we assume a control node \( n_i \) may have \( \alpha_1 c_i \) vulnerabilities, where \( \alpha_1 \) is the vulnerability density coefficient [3]. In the worst case scenario, where a vulnerability allows an attacking packet to alter the control flow to any code address, let \( \alpha_2 c_i \) denote the amount of the code of a control node \( n_j \) that may be utilized in exploitation, where \( \alpha_2 \) is a utilization probability coefficient. Hence, the security of the control flow between \( n_i \) and \( n_j \) can be quantified as \( \beta_{i,j} = \beta_{i,j} \alpha_1 \alpha_2 c_i c_j \), where \( \beta_{i,j} \) is the ability of attackers to alter the control flow from \( n_i \) to \( n_j \) if \( n_i \) has a vulnerability.

When the code layout of a program is fixed, \( \beta_{i,j} = 1 \) for all \( j \), assuming attackers have the ability to alter the control flow to any \( n_j \). Hence, the security of \( n_i \) is \( S_i = \sum_{i=0}^{N-1} s_{i,j} = \alpha_1 \alpha_2 C_i^2 \), where \( C = \sum_{j=0}^{N-1} c_j \). The overall security of the program is thus \( S_T = \sum_{i=0}^{N-1} S_i = \alpha_1 \alpha_2 C^2 \).

When the code layout of a program is randomized, \( \beta_{i,j} \approx 0 \) for all \( j \neq i \), since attackers do not know the address of any \( n_j \) in a remote sensor. However, \( \beta_{i,i} \approx 1 \), if assuming the vulnerability in \( n_i \) allows attackers to use relative addressing for exploitation. Hence, the security of \( n_i \) is \( S_i = \sum_{i=0}^{N-1} s_{i,j} = \alpha_1 \alpha_2 C_i^2 \), and the overall security of the randomized program is \( S_T = \sum_{i=0}^{N-1} S_i = \alpha_1 \alpha_2 \sum_{i=0}^{N-1} C_i^2 \). Therefore, the security that the randomization adds into a program can be quantified as:

\[
\text{sec} = 1 - \frac{S_T}{S_I} = 1 - \frac{\sum_{i=0}^{N-1} C_i^2}{C^2}.
\]

In one extreme scenario where each control node contains only one instruction, \( \text{sec} = 1 - \frac{1}{C} \approx 1 \), which means it is almost impossible to exploit unless an instruction can exploit itself. In the other extreme scenario where a program has only one control node, \( \text{sec} = 1 - 1 = 0 \), because no randomization is introduced into the code image. Note that \( \text{sec} = 0 \) does not imply that a program is sure to be exploitable. Rather, it only means if a program has a vulnerability, it can be exploited, because no extra protection is applied to it.

Randomizing the code layout also requires recomputing ITAs and ETAs for reusing the executable in a sensor. Hence, they are one of the main factors determining the overhead of diversification. As discussed in Section 3.3, the ITAs only incur a little amount of overhead in the recomputation. The overhead is mainly determined by the ETAs, including the transmission of the ETA table that needs to be disseminated with the code to all sensors and the recomputation of the ETAs that is needed to obtain their values in the final executable. Hence, reducing the number of ETAs is the key to reducing the overhead of diversification.

Figure 4 shows that the number of ETAs in a DG can be reduced by merging control nodes. In the figure, the number marked on an edge between two control nodes is the count of the ETAs that point from the source node to the destination node. In Figure 4(a), the total number of ETAs in the DG is 14. After merging as in Figure 4(b), the number of ETAs is reduced to 5. However, the merging sacrifices security. The more nodes that are merged, the more it becomes possible to exploit a vulnerability. The trade-off between overhead and security can be determined by various approaches of restructuring the DG, which is not the focus of this paper.

### 4. PERFORMANCE ANALYSIS

The proposed code diversification scheme has been implemented in Deluge of TinyOS version 2.1 for Micaz motes. A Micaz mote has a 4KByte SRAM for data, a 4KByte EEPROM, and a 256KByte flash memory. The flash memory is divided into 4 flash volumes, and each volume can store a code image of up to 64K bytes. Hence, a sensor can hold up to 4 programs in its flash memory. Our implementation reserves the GoldenImage volume to store the diversified executable. The code images stored in the other three flash volumes are disseminated using the same Deluge protocol.

In Deluge, a sensor issues the programImageAndReboot command to the NetProgC component, which then programs the program memory with the specified program stored in a flash volume and reboots the sensor. Our implementation adds a new component ReProgC in Deluge as a wrapper of the NetProgC component to intercept the programImageAndReboot command. Upon receiving the programImageAndReboot command, the ReProgC component retrieves the code image stored in the specified flash volume and diversifies and stores the code in the GoldenImage volume. Then, it invokes the NetProgC component to reprogram the memory component to boot the sensor using the diversified executable stored in the GoldenImage volume. All intermediate data for diversification is stored in the sensor’s EEPROM.

The overhead of the CD scheme includes the transmission cost for disseminating the code image to sensors and the computational cost for diversifying the executable in sensors. The overhead of the CD scheme results in extra energy consumption and time delay in network reprogramming.

#### 4.1 Transmission Cost

As the CD scheme does not change the underlying networking protocol used by Deluge, its transmission cost is mainly due to the extra code and data produced by the CD scheme. Table 1 shows the code sizes of three programs built with or without the CD scheme.
scheme. The three programs are the Deluge test suites in TinyOS. The “GoldenImage” program is simply made of the core component of Deluge without any other functions. The other two programs have some applications in addition to Deluge. As the CD scheme may share some code with Deluge and applications, the actual amount of extra code varies by the applications. The table shows that Deluge itself has about 33K bytes of code, and the CD scheme adds about 2.70K to 3.05K bytes of extra code, which is about 9% of Deluge. Note that the extra code of the CD scheme is almost a constant overhead and not affected by the size of sensor applications.

Table 2 shows the sizes of the three components in the CD header. The size of the meta data is constant at 16 bytes. The size of the node table is proportional to the number of control nodes, but limited to 510 bytes. Each entry in the node table is an address of two bytes. The size of the ETA table is proportional to the number of unique ETAs, which is determined by the DG of an application as discussed in Section 3.4. As shown in Figure 3, each entry in the ETA table has a node ID and an address, which takes three bytes in total. If an application has fewer than 1000 unique ETAs in its DG, the total size of the diversification header is smaller than 4K bytes, which can be stored in the EEPROM of a Micaz mote for diversification. This is quite achievable, since a 33KByte program has fewer than 300 unique ETAs and Deluge can support only up to 64KByte programs.

### 4.2 Computational Cost

The CD scheme incurs extra computational cost due to diversifying the code and storing the diversified code into the GoldenImage volume, in addition to the cost of Deluge for storing a code image into a flash volume. This extra cost is the sum of the following computational items:

(a) Copying the CD header to the EEPROM,
(b) Copying the code to the GoldenImage volume,
(c) Rebuilding the Deluge header for the diversified executable,
(d) Randomizing the node table,
(e) Recomputing ETAs and ITAs.

Item (a) is mainly determined by the size of the CD header, which has been analyzed in the previous section and is bounded by the size of the EEPROM. Item (b) is equivalent to copying a code image from one flash volume to another flash volume. Item (c) mainly involves reading code pages from the GoldenImage volume, computing their CRCs, and storing CRCs into the Deluge header. The code in a flash volume may produce up to 64 CRCs. The latter two cost more than the first one, as the code size is much larger than the size of the CD header and the data access of flash memory takes more time and consumes more power. Hence, the cost of the first three items is largely determined by the size of the code image, which cannot be reduced by optimizing the DG of a sensor application. The cost of item (d) is \( O(N) \) of accessing the node table in EEPROM, where \( N \) is the number of control nodes.

In item (e), the computation for the ITAs can be ignored compared with the computation for the ETAs. As shown in Table 3, the number of ITAs is much smaller than the number of ETAs. In addition, the computation of ITAs only includes recomputing their values in the diversified executable in CPU, which costs much less than accessing EEPROM or flash memory whether in terms of time or energy consumption. Note that the cost of reading and writing ITAs and ETAs in the flash memory has already been counted in item (b). Therefore, among all the cost items, only the computation for ETAs in item (e) is a cost that can be reduced by restructuring the DG of an application.

As discussed in Section 3.4, the computation for ETAs is determined by the number of ETAs in the DG of an application, and reducing this cost will also reduce the security brought by code diversification. However, the rates of reducing overhead and security could be different with different merging strategies. Use the Blink application as an example. Its DG shows that most nodes that are large with many outgoing ETAs have few incoming ETAs, while most nodes that are small with few outgoing ETAs have many incoming ETAs. Based on this characteristic, merging nodes with many incoming ETAs first can reduce overhead at a higher rate than security, since their incoming ETAs will be turned into ITAs and they do not have many outgoing ETAs. Figure 5 confirms this merging strategy. Even when the overhead is reduced to 60% of the DG with no merged nodes, security is only reduced about 10%. When the overhead is reduced to less than 60%, the security and the overhead are proportionally reduced as the remaining nodes have similar code size and ETAs. When all nodes are merged into one, both security and overhead reach 0%.

### 5. RELATED WORK

The proposed code diversification scheme is closely related to research on software diversification, secure network reprogramming, and modular reprogramming.

Software diversification has been applied to protect various computer systems. One trend of diversification is code obfuscation that targets making code hard to understand. Various approaches [8, 14] have been developed to transform the control flow of a program to hide the real procedure of execution and the intent of the program. However, to protect a sensor network, it requires all sensors to be able to obfuscate their own code, which needs a tool to analyze their own control flow and data flow. Such a tool will incur a large
overhead to sensors.

The other trend of diversification is randomization, which changes the layout of code and data of a program [4]. Because exploiting a vulnerability normally requires the knowledge of the structure of code and data, randomizing a running program makes the structure unpredictable to attackers. The proposed CD scheme adopts this approach as it is efficient to sensors. However, sensors receive the same code image disseminated in network reprogramming. The scheme studies how to make sensors extract information carried in a code image to randomize their executables.

Secure network reprogramming in sensor networks has drawn much attention recently. Two major issues of secure network reprogramming were addressed. One is the security of the data plane in the code dissemination of network reprogramming [6, 16]. As a code image could be changed, forged, eavesdropped, or denied by malicious sensors, various schemes have been developed to provide integrity and confidentiality on the code image and also ensure DoS resilience in code dissemination. The other issue of secure network reprogramming is the security of the control plane in code dissemination [15]. As network reprogramming systems issue commands to sensors to manage code images, these commands are critical to securing code dissemination. Forged commands can deceive sensors to erase their stored code images or reboot themselves to a wrong code image. Lightweight authentication mechanisms are introduced into the management commands.

This paper discusses another aspect of network reprogramming. This paper focuses on how to allow sensors to diversify their executables from a code image received in network reprogramming. The proposed CD scheme does not change network reprogramming protocols, but rather focuses on what kind of information should be carried in a disseminated code image. The proposed CD scheme is complementary to the schemes for securing network reprogramming in sensor networks.

Modular reprogramming has also been developed for software update in sensors [10]. It uses loadable modules to build a sensor system. When a module needs to be updated, only the module will be reprogrammed. Such a system can be easily extended to support dynamic linking and loading [5]. As in a regular computer, a dynamic linker can be implemented in sensors. If a module needs to be updated, a compact ELF file of the module will be distributed. The file contains a symbol table and other information for the dynamic linker to relocate and link all symbols. Hence, the code and data layout of the modules can be diversified.

However, all the modular reprogramming schemes need a kernel to manage the modules. The kernel cannot be reprogrammed. Hence, such reprogramming is not full image reprogramming, and diversification based on such software architecture will exclude the kernel. As previous research has shown that any code in a sensor could be exploited, the proposed CD scheme targets full image diversification that includes randomizing all functions and interrupts of applications and TinyOS.

6. CONCLUSIONS

This paper presented an efficient code diversification scheme for network reprogramming in sensor networks. The scheme uses Deluge to disseminate code images of sensor applications with diversification information and allows sensors to randomize the layout of their own executables. Such diversification can defeat a wide range of attacks that exploit the knowledge of code layout, as no sensors have the same code layout in their executables. The proposed scheme adds about 9% extra data and code to the code image of Deluge. This paper showed that the security and the overhead of the proposed scheme can be determined by the construction of the diversification graph of a sensor application. Except the cost determined by the code size, the overhead can be reduced to 60%, while sacrificing only 10% of security. In the future, more diversification schemes on data and code flow will be investigated for network reprogramming.

7. REFERENCES